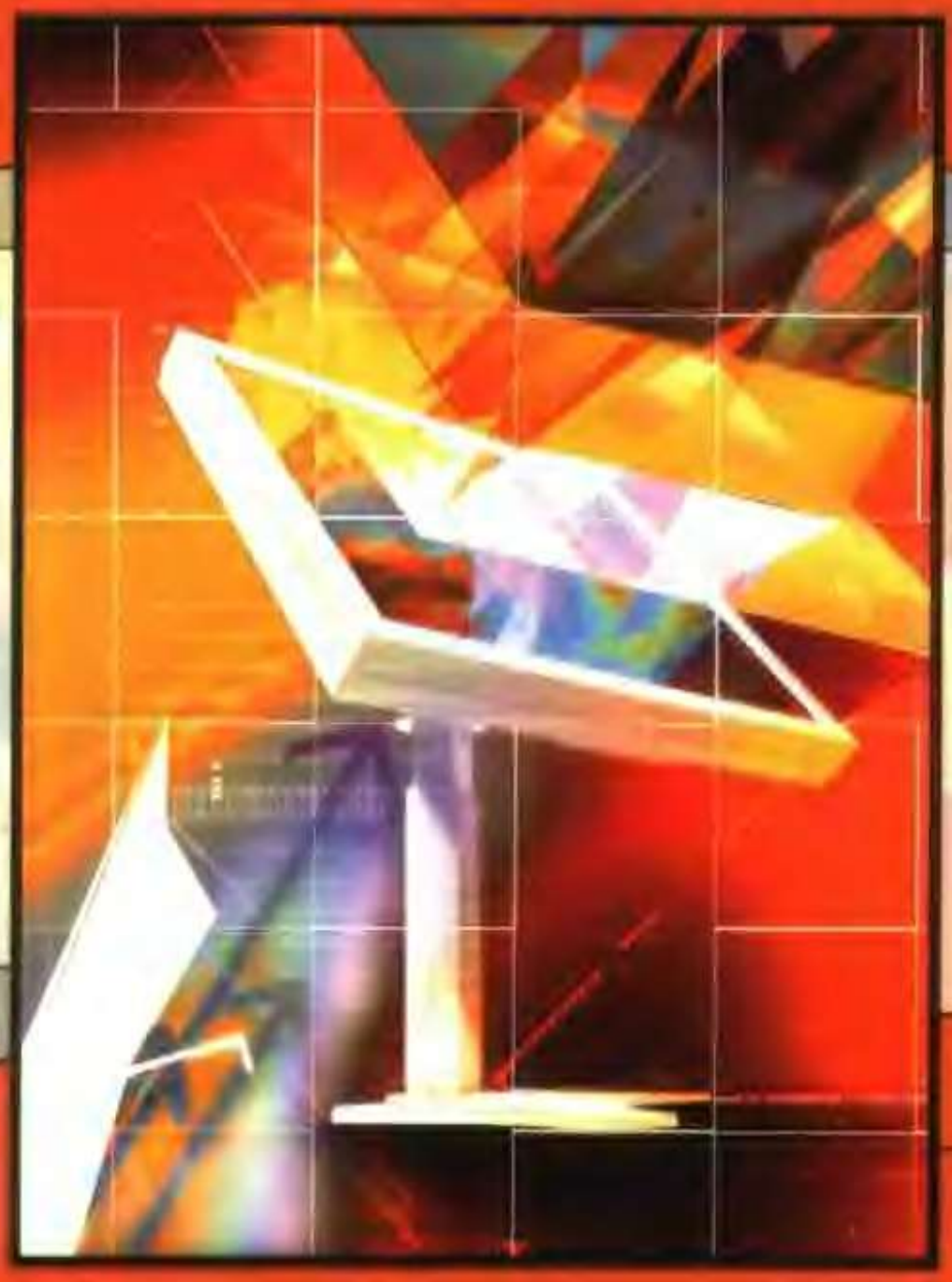
Borland In-Depth Series \ Borland 大系

Broadview

Delphi 源代码分析



周爱民 著

CD-ROM included

電子工業出版社
PUBLISHING HOUSE OF ELECTRONIC INDUSTRY
http://www.phei.com.cn

Broadview

Borland In-Depth Series \ Borland 大系

博文视点资讯有限公司总策划

Delphi源代码分析

内容提要



本书通过对Delphi内核(RTL)源代码进行分析，深入阐述了Delphi内核(RTL)的原理及其实现。全书从Nico Bendlin编写的著名最小化内核示例程序MiniDExe开始讲起，基于MiniDExe分析Delphi在编译器一级的技术内幕，带领读者一窥Delphi的奥秘。随后作者基于这个内核逐层地包装代码，将Delphi的各种功能的具体实现一层层地，通过剖析关键核代码并进行反编译分析的方式，全面分析各种操作系统运行环境机制在源代码中的实现。全书内容详实，阐述精辟，深入，主要包括Delphi的编译器在Windows，Delphi RTL和用户代码之间的交互；Delphi RTL内核代码的完整实现；与Delphi内核相关的操作系统机制：初始化入口代码，模块，内存，线程，资源，异常处理机制等。是一本不可多得的高端技术图书，适合中、高级Delphi开发人员阅读。

作者简介



周爱民（Aimingoo） Delphibbs（大富翁论坛）资深成员

◆ 1999～2000年为AV95的三个核心开发人员之一，独立完成了该核心底代码编写。

◆ 2000～2002年完成极光网络数据库中心系统，荣获河南省高新技术产品二等奖。

◆ 2003年被美国Borland公司授予“Borland Delphi产品专家”称号，同时授予“论坛特别贡献奖”，为唯一同时被授两个奖项的人员。

◆ 2003年11月，受邀参加Borland公司成立20周年举办的“Borland首届开发者大会”，发表题为“利用Delphi实现在Microsoft.NET Framework下的开发”的演讲。

作者主页 http://www.doany.net

联系方式 aim@263.net

博文视点书友论坛：http://forum.broadview.com.cn

网上订购：http://www.dearbook.com.cn 第二书店 第一服务

ISBN 7-121-00303-1

9 787121 003035 >

责任编辑：周 筠 张兴田

责任校对：陈元玉

版式设计：张兴田

封面设计：方 舟 张兴田

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 7-121-00303-1 定价：40.00元(含光盘1张)

Borland In-Depth Series\Borland 大系

Delphi 源代码分析

周爱民　著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书通过对 Delphi 内核(RTL)源代码进行分析，深入阐述了 Delphi 内核(RTL)的原理及其实现。全书从 Nico Bendlin 编写的著名最小化内核示例程序 MiniDExe 讲起，基于 MiniDExe 分析 Delphi 在编译器一级的技术内幕，带领读者一窥 Delphi 的核心。随后作者基于这个内核逐层地包装代码，将 Delphi 的各种功能的具体实现一一展现，通过列出关键性代码并进行系统性分析的方式，全面分析对象结构、VCL 和 COM 等在源代码中的实现。全书内容详实，阐述精辟、深入，主要议题包括：Delphi 的编译器在 Windows、Delphi RTL 和用户代码之间的交互；Delphi RTL 内核代码的完整实现；与 Delphi 内核相关的操作系统机制；初始化入口代码、模块、内存、线程、资源、异常处理机制等。

本书是一本不可多得的高端技术图书，适合中、高级 Delphi 开发人员研读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容
版权所有，侵权必究

图书在版编目(CIP)数据

Delphi 源代码分析 / 周爱民著. —北京：电子工业出版社，2004.9
（Borland In-Depth Series\Borland 大系）

ISBN 7-121-00303-1

Ⅰ.D... Ⅱ.周... Ⅲ.软件工具—程序设计—代码—程序分析 Ⅳ.TP311.56

中国版本图书馆 CIP 数据核字（2004）第 088925 号

责任编辑：周 筠 张兴旭
责任校对：陈元玉
印 刷：北京智力达印刷公司印刷
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
经 销：各地新华书店
开 本：787×980 1/16 印张：23.5 字数：550 千字
印 次：2004 年 9 月第 1 次印刷
印 数：5 000 册 定价：40.00 元（含光盘 1 张）

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn

序

第一次和 Aimingoo 见面是 2003 年底 Borland Conference 2003 在北京举行之际，Aimingoo 宏亮的声音、诚恳的态度立刻引发了我的好感。在 BorCon 排练时 Aimingoo 努力不懈，一直坚持到最后，我当时心中就想，这位年轻人真是不错。到了 BorCon 结束之后，我和 Aimingoo 有更多机会可以聊天，让我了解了 Aimingoo 的专长领域，也很佩服 Aimingoo 这么年轻就拥有这么多的开发经验。

在知道 Aimingoo 要写有关 Delphi Run Time Library 的书时，心中又惊又喜。惊的是这可是一个浩大的工程，喜的则是 Aimingoo 愿意花时间写这方面的书籍，造福想要一窥究竟的 Delphi 使用者。当 Aimingoo 把他的书籍目录和初稿给我过目时，我又不禁羡慕他的成果以及 Aimingoo 的书籍和拙作《Inside VCL》是如此地相辅相成，Aimingoo 的书籍叙述了驱动 Delphi 应用程序的引擎动力，《Inside VCL》则说明了 Delphi 应用程序使用的系统框架。

Aimingoo 请我为他的新书写序实在是我的荣幸，除了他丰富的知识和精湛的技术之外，Aimingoo 做事的态度更令人敬佩。结合技术和努力的结晶，Aimingoo 为广大的 Delphi 使用者写出了另一本 Delphi 领域的著作，对于想要深入了解 Delphi 执行时期运作细节的 Delphi 使用者而言，这是值得阅读和珍藏的一本好书。

最后我个人也要谢谢 Aimingoo 又为**李维工作室**产生了一本高质量的技术书籍，让**李维工作室**系列书籍更为丰富和充实。希望 Aimingoo 能够继续写出好书，不断帮助我们这群 Delphi 使用者更上一层楼。

李维 2004/3

前　言

关于本书

在用 Delphi1.0 时，我便开始阅读 Delphi 的源代码了。大约是在五年前，我曾试图做一个名为“Delphi 源代码深入分析”的网站，后来终于放弃了这个计划。而自此，写现在这样一本书的想法便积蓄至今。

真正激发我做这样一件事的，是 Nico Bendlin 写的 MiniDExe。这是我所见过的用 Delphi 写的最小版本的可执行程序，它使我得以一窥 Delphi 的核心。于是，我开始基于 MiniDExe 分析 Delphi 在编译器一级上的真相。随后，我分析了对象结构、VCL 和 COM 等在源代码中的实现。至此，本书的基本知识框架组织完成。

本书的知识结构

本书以内核的原理及其实现为主，主要讲述：

- Delphi 的编译器在 Windows、Delphi RTL 和用户代码之间的交互；
- Delphi RTL 内核代码的完整实现；
- 与 Delphi 内核相关的操作系统机制；
- 初始(入口)代码、模块、内存、线程、资源、异常处理机制等。

所有在本书中使用到的术语、参考资料等将以附录的形式列出。

本书基于 Delphi 7.0 所提供的源程序分析。

什么是“内核”

通常意义上的开发工具“内核”，指的是 RTL(RunTime Library)。在 Delphi 中，RTL 是指源码的*$(Source)\RTL* 目录中的全部代码。但以纯粹的 RTL 的概念来理解的话，事实上它应当是指 *System.pas* 和 *SysInit.pas* 中的全部代码(包括 *GetMem.inc*)——这甚至不包括 *SysUtils.pas* 这个单元。

在本书中所讨论的内核，基本上是指*$(Source)\RTL\Sys* 目录中的代码。此外，还包括：

- *$(Source)\RTL\Win*：*Windows.pas* 的部分代码。
- *$(Source)\RTL\Common*：*Classes.pas* 和 *SyncObjs.pas* 中的部分代码。

如何阅读这本书

写这本书的过程中，我几乎无时不在问自己“Delphi 如何实现这个”或者“Delphi 如何实现那个”，所以这本书在很大程度上都是在描述“Delphi 如何做到”，而不是“用 Delphi 如何做”。因此，我建议你以技术探索的角度来读这本书。你不要指望在本书中发现太多能直接在工程中使用的技术与技巧。

本书虽然是立足于 Delphi 源代码分析，但书中并不会大段地给出 Delphi 产品中的源码。因为 Borland 并没有这样的授权。通常情况下，我会列出关键性的代码，并进行系统的分析。因此，如果需要，读者可能必须自行对照源码中的.*pas* 文件来阅读一些章节。

本书不包含内核源码的哪些内容

本书立足于对 Delphi 内核(RTL)源代码的深入分析，但为了叙述的必要，仍剔除了部分内容。这些内容主要是指：

一、 BCB 兼容代码

我不是太了解 BCB。因此除非必要，我不会在本书的章节中讨论与 BCB 相关的内容。基于同样的原因，在 Delphi 内核源码中与 BCB 相关的(绝大多数)代码也被我忽略了。

二、 调试代码

除了在讲述异常机制时，其它绝大多数情况下我都不会讨论在 Delphi 内核源码中与调试相关的代码。那的确是相当复杂的一套代码。不过在读完本书之后，我相信读者已经有能力自行开始分析这些代码了。

三、 面向 Linux 的移植：CLX

要讨论 CLX，则必须进一步深入了解 Kylix 和 Linux 操作系统。如果在本书中以大量篇幅出现这些内容，将会降低本书在主流的桌面开发平台(当然，你知道我指的是什么)上的实用性，同时，也将使关于 RTL 的分析显得混乱。

本书的示例源码

我绝不想用大量源码来充斥书页。因此，本书中大多数地方的源码引用只是点到即止。很多时候，你会看类似于“...”的省略。或许有部分读者会认为在光盘中(而不是书中)阅读源码是件痛苦的事，但是我相信，如果与使用数页甚至十数页的源码来充填内容、增加厚度相比，将这些东西放在光盘中还是更对得起读者的钱包的☺

两本书

这里提到两本书。提及这两本书，是因为在底层以及原理的分析方面，这两本书当属扛鼎之作。因此本书会在内容上尽量避免与它们重复。

其一是 Ray Lischner 著、中国电力出版社出版的《Delphi 技术手册》。这本书基于 Delphi 5，用 130 页的内容概述了 Delphi 的基本语言特征，其后的 500 页则是一个完整的语言参考。基本语言特征部分言简意赅但直指核心，句句发人深省；语言参考部分十分详尽，且附有精彩的示例。

其二是李维先生著、电子工业出版社出的《Inside VCL》。

鉴于《Inside VCL》一书卷帙浩繁，解析深湛，因此本书剔除了与 VCL 部分有关的内容(原定计划中是有的)，至于 OOP 在 Delphi 中的实现以及可能的一些 VCL 和 COM 方面的补遗，我会放在其它书中单独去讲。而《Delphi 技术手册》一书中对 RTL 的过程、函数描述详尽，因此本文在这些例程的功能方面，也不多费笔墨。

.NET 相关问题

本书不讲.NET Framework。在绝大多数读者看来，.NET 的出现，意味着 Delphi 的终结，很多朋友甚至对写这样一本 Delphi 内核分析的书的必要性提出质疑。然而在我看来，即便是.NET，在内核结构上仍与 Win32 内核存在千丝万缕的联系，更何况.NET 中的非托管代码，还需要用 Win32 原生应用开发工具来实现。所以 Delphi 7.x 与 Delphi .NET 是两条不同的产品线，也有着不同的发展方向。

即使是在.NET 大行其道的时候，Win32 开发以及 Win32 应用的维护仍然要持续很长一段时间，正如至今还有人在用汇编写驱动程序一样。

免责声明

本书中许多内容都是未见于 Delphi 的标准文档的。这意味着某些结论是非官方的，在更新(或者更旧)的版本中，不会得到认可和支持。我只是从源代码中得到了这些结论，并通过实例来证实了它们。如果你需要在其他版本的 Delphi 中实现相应的功能，你应当重新编译并运行测试程序。

如非必须，你不应当在商业软件中使用某些技巧。这不是授权问题，而是安全性的问题。通常，如果你发现“必须使用技巧来解决问题”，那么你应当考虑“是否方案设计错误”。我建议你在商业化软件中应用这技巧时，应当做出详尽的源码注释，并在项目中添加一个专用的测试程序。

我保证在本书中的所有结论和技巧都经过验证和测试。但如果读者未经过进一步的测试而使用它们，所导致的问题，是我所不负责任的。

随书光盘

本书附随书光盘一张，包含了书中所有示例程序源代码以及相关工具。光盘基本目录结构如下：

- *\Book*：本书各章节中所使用的示例程序的完整源代码。
- *\third*：本书在正文中介绍过的第三方代码/开发项目。
- *\tool*：本书编写过程中使用过的一些分析工具。

在绝大多数目录下都将包含一个文件“说明.txt”，该文件描述了该目录中的代码或工具的

相关资料。如果本目录下未包含该文件，则相关的说明应当在上一级目录的“说明.txt”文件中。

在本书正文中，如果必须指明光盘中的文件位置，通常以如下格式给出(其中$(CD)表示光盘所在驱动器)：

```
$(CD)\third\chapter 1\MiniDExe.zip
```

本书所附的光盘中不带有任何 Delphi 产品光盘中的源代码。但是，部分程序会用到这些源代码，这需要读者自行提供——它们应在你的 Delphi 安装目录中。[①]

排版格式与惯例

通常情况下，我不会在书的正文部分包含注释，而会写成脚注。仅在不影响阅读的情况下，我会在正文中加入由“()”包含的可省略语言，通常它们是用于强调、限定或者极少量的注释。

在书中包含的源代码中存在大量的“//...”，这些被省略掉的部分可能存在于 Delphi 自己的源代码或者随书光盘中。这取决于你正在阅读的是哪一部分代码。但在极少数的情况下，它们表明是在前文中出现过的代码，这时我通常会补充一行注释，表明你可以在什么位置看到这些代码。例如：

```
// ...
// (参见上例)
```

在源代码的第一行可能出现一个单行的注释。这个注释表明其后的代码引用自哪个单元，以及该单元中的哪个例程(如果随后的源码中没有给出例程名字)。但是，如果源代码出自 *System.pas* 单元，这一行注释将不会存在。此外，这个单行注释还有一种特殊的情况：如果随后的代码是改写自 Delphi 源代码而非直接复制，那么注释将以“code referenced by”开始，而不是通常的“code form”。例如：

```
// code referenced by System.pas, _InitResStrings()

TResStringInitItem = record
  variableAddress: String;
  resStringAddress: PResStringRec;
end;
```

① 请注意这些源程序只包含在某些版本中，如果你的 Delphi 中不包含它们，请安装更高级别的商业版本。

一些代码并不是.*pas*中的源代码，而是从 Delphi IDE 的 CPU 窗口中复制出来的汇编代码[①]。通常这些代码会包括源代码行、运行期地址、二进制码和反汇编代码行。这些代码以类似如下的形式提供：

```
Project1.dpr.5: ShowMessage('Test');
00451F6C B8841F4500      mov eax,$00451f84
00451F71 E8D2FBFFFF      call ShowMessage
```

很多时候，我习惯于使用 DOS 命令行来做一些事情(事实上，本书中的绝大多数代码都是我用 *Notepad.exe* 书写并用 DCC32.EXE 编译的)，因此本书中会有一些 DOS 命令行。为了区分它们，我使用了加粗的文字，而且它们通常将以类似于“*C:\>*”这样的形式开始。而紧接其后的内容，则是执行该命令行后的屏幕显示(可能是部分的显示结果)。例如：

```
C:\>tdump E:\Delphi7\Bin\tregsvr.exe
#   Name      VirtSize   RVA      PhysSize  Phys off  Flags
--  --------  --------  --------  --------  --------  --------
08  .rsrc     00001800  00018000  00001800  00015A00  50000040 [IRS]
09  .debug    000E1AAC  0001D000  000E1AAC  00017200  50000040 [IRS]
```

另外，对于本书中出现的英文，采用了不同的字体表达不同的概念，概列如下：

- ✧ 正文中一般的英文采用 Times New Roman
- ✧ 行文中的代码采用 Lucida Sans Typewriter
- ✧ 大段代码采用 Courier New 字体
- ✧ 代码中的注释采用楷体_GB2312 字体
- ✧ 具有特定含义的概念或人名采用 Arial 字体
- ✧ 超级链接采用 Arial Narrow
- ✧ 正文中文件名/磁盘路径采用 *Arial* 斜体

致　谢

我当然得感谢 Nico Bendlin(nicode@gmx.net)，尽管我从未见过、也没有联系过他，但是如果没有他的 MiniDExe，我可能永远也不能开始写这本书。

① 在随书光盘中提供了一个笔者编写的 IDE 专家工具 CpuWHelper，可以方便地从 CPU 窗口中复制汇编代码。

我得感谢那个著名的 Delphi 大富翁论坛(http://www.delphibbs.com/)。感谢 yysun 创建了这个论坛，感谢 Soul 近一年来不辞辛劳地无偿维护着这个论坛，并以本书的出版纪念五年前离开论坛的 dwwang。在此也感谢活跃其中的诸多朋友，我的工作离不开大家的支持和帮助。谢谢。

我得感谢 CSDN 与电子工业出版社的朋友，这包括蒋涛、周筠、韩磊、方舟、张兴田，以及所有关注本书的编写、出版、发行的朋友。感谢我的老朋友王寒松先生向蒋涛推荐了本书，尽管那时这本书还没有写到 100 页。

感谢李维先生在百忙之中为本书作序。他对我的赞誉之词既令我深感荣幸亦令我不胜惶恐，以致于作每段文字时都倍加小心，生怕出现任何疏漏差错，而有负于李维先生对我的信任和对本书的推介。

我还得感谢我身边的她(Joy Xie)。尽管经过长达数年之久的斗争，她仍然未能迫使我改掉昼伏夜出的作息习惯。但是我得承认，如果没有她，我无法写完这本书(中途死于饥饿或本书终结于我觅食的过程中)。此外，在写这本书的过程中，我们结婚了☺

感谢给予我支持和帮助的所有朋友！

目　录

序 (i)

前言 (I)

第一部分　Delphi 内核深入剖析（I） (1)

第 1 章　最小化 Delphi 内核 (3)

1.1　MiniDExe 如何实现内核最小化 (3)

1.1.1　MiniDExe 中的 System.pas 单元 (4)

1.1.2　MiniDExe 中的 SysInit.pas 单元 (5)

1.1.3　MiniDExe 中的项目文件 MiniDExe.dpr (6)

1.2　一些其他的内核优化 (6)

1.3　为什么要研究最小化内核 (7)

第 2 章　基本数据类型的实现 (9)

2.1　基本数据类型 (9)

2.2　变量与常量 (11)

2.2.1　全局变量与局部变量 (11)

2.2.2　动态分配的内存 (12)

2.2.3　换一个方式来理解 (13)

2.2.4　常量 (14)

2.3　数据结构的实现 (16)

2.3.1　简单类型 (16)

2.3.2　字符串 (16)

2.3.3　构造类型 (26)

2.3.4 指针类型 …… (28)
2.3.5 过程类型 …… (28)
2.4 数据结构相关的例程 …… (29)
2.4.1 标准 Pascal 的内置例程 …… (30)
2.4.2 字符串操作例程 …… (31)
2.5 变量的类型检测与强制转换 …… (47)
2.6 引用 - 计数 - 写复制与类型信息 …… (48)
2.6.1 引用计数与增加引用 …… (48)
2.6.2 “增加引用”何时发生 …… (49)
2.6.3 增加引用的操作是依赖类型信息来实现的 …… (51)
2.6.4 写复制与值参数的备份 …… (53)
第 3 章 BASM(Borland 汇编语言)精要 …… (55)
3.1 BASM 概念简要 …… (55)
3.2 表达式的类别与类型 …… (56)
3.3 数据定义和数据类型强制转换 …… (57)
3.4 例程入口参数及调用约定 …… (61)
3.5 例程和 API 的调用与流程控制 …… (62)
3.6 完全汇编例程与内嵌汇编例程 …… (65)
3.7 汇编例程中的返回值约定 …… (66)
3.8 其他 …… (67)
第 4 章 初始化与结束化过程 …… (71)
4.1 变量的初始化与结束化 …… (71)
4.1.1 初始化的必要性 …… (72)
4.1.2 如何初始化 …… (73)
4.1.3 如何结束化 …… (74)
4.2 例程的初始化与结束化 …… (76)
4.3 单元初始化与结束化 …… (77)
4.3.1 单元初始化与结束化的内部例程 …… (77)
4.3.2 其他初始化例程 …… (79)
4.4 模块初始化与结束化 …… (79)

4.4.1 模块入口代码 …… (79)
4.4.2 编译器决定的程序执行流程 …… (80)
第 5 章　面向 Windows 开发的基本实现 …… (83)
5.1 Win32 应用程序：EXE …… (83)
5.1.1 适应 Win32 应用程序的最简化内核 …… (83)
5.1.2 初始化例程_InitExe() …… (85)
5.1.3 内部模块表管理例程 …… (86)
5.1.4 .EXE 启动例程_StartExe() …… (87)
5.1.5 应用程序的结束化控制 …… (88)
5.2 32 位的 DOS：控制台应用程序 …… (91)
5.2.1 控制台应用程序的模块入口代码 …… (91)
5.2.2 控制台应用程序的最小化实现 …… (92)
5.2.3 控制台应用程序的 Delphi 实现 …… (93)
5.2.4 文件操作例程与控制台应用程序 …… (94)
5.2.5 控制台的开启与关闭 …… (97)
5.2.6 CRT 单元与 Input、Output 的重载 …… (98)
5.3 动态链接库：DLL …… (99)
5.3.1 丢失的 DllMain() …… (99)
5.3.2 _InitLib()例程 …… (101)
5.3.3 _StartLib()例程 …… (101)
5.3.4 .DLL 的结束化过程 …… (102)
5.3.5 DllProc 与 DllMain()的不同 …… (105)
5.3.6 动态链接库的内核最小化 …… (106)
5.4 Delphi 的动态链接库：包 …… (108)
5.4.1 包的主要规则 …… (108)
5.4.2 Delphi 中的包与普通 DLL 的区别 …… (110)
5.4.3 包的 DllMain() …… (111)
5.4.4 包的载入例程 LoadPackage() …… (112)
5.4.5 真正的初始化例程 InitializePackage() …… (112)
5.4.6 包的卸载例程 UnloadPackage() …… (115)
5.4.7 包的基本输出例程 …… (115)
5.4.8 内部例程_PackageLoad()与_PackageUnload() …… (116)

5.4.9 包的内核最小化 ……(117)
5.5 其他 ……(118)
5.5.1 初始化上下文中 OuterContext 域的使用 ……(118)
5.5.2 入口代码中的堆栈使用深入分析(内存现场) ……(119)
5.5.3 再论入口代码 ……(123)
第 6 章 Delphi 的积木艺术(PE) ……(127)
6.1 PE 文件结构概要 ……(127)
6.1.1 文件头 ……(129)
6.1.2 节表 ……(131)
6.1.3 节 ……(132)
6.1.4 PE 文件与内存映射 ……(134)
6.1.5 有关相对虚地址的计算 ……(138)
6.2 Delphi 的 PE 文件头中一些重要的域 ……(142)
6.3 Delphi 的 PE 文件中一些重要的节 ……(144)
6.3.1 线程局部存储(.tls 和.rdata) ……(144)
6.3.2 资源节(.rsrc) ……(145)
6.3.3 导入、导出表(.idata 和.edata) ……(146)
6.3.4 数据节与代码节(DATA、BSS 和 CODE) ……(154)
6.3.5 重定位节(.reloc) ……(157)
6.4 3.5K 的秘密 ……(159)
6.4.1 Delphi 的 PE 文件头部 ……(159)
6.4.2 Delphi 的 PE 文件的节及其默认对齐 ……(160)
6.4.3 还可能更小吗 ……(160)
6.4.4 3.5K 代码的内存映射 ……(161)
6.5 入口点 ……(162)
6.5.1 磁盘文件上的入口代码(RAW Address) ……(163)
6.5.2 反编译器使用相对基地址定位的入口代码(RVA) ……(164)
6.5.3 载入到内存之后(运行期)的入口代码(VA) ……(164)
第二部分 Delphi 内核深入剖析（II） ……(167)
第 7 章 Delphi 的内存管理器 ……(169)
7.1 Delphi 的内存管理器实现框架 ……(169)

7.2 内存页管理 …… (171)
7.3 堆 …… (172)
7.4 MemoryManager 及相关例程 …… (172)
7.5 GetMem.inc 中的重要例程 …… (174)
7.5.1 堆块及其管理例程 …… (175)
7.5.2 虚地址空间(Address space)管理 …… (178)
7.5.3 已提交的内存空间(Committed space)管理 …… (179)
7.5.4 用户调用例程(actually calls)的实现 …… (180)
7.5.5 初始化、结束化与其他辅助例程 …… (194)
7.6 遍历全部内存块 …… (195)
7.7 共享内存管理器 …… (197)
7.8 第三方内存管理器 …… (200)
7.9 小结 …… (201)
第 8 章 错误和异常 …… (203)
8.1 错误 …… (203)
8.2 断言 …… (206)
8.3 Windows 与 Delphi 中的异常处理机制概要 …… (209)
8.4 编译器对异常处理机制的实现 …… (212)
8.4.1 最小化内核的启示 …… (212)
8.4.2 从操作系统的角度来理解编译器行为 …… (216)
8.4.3 try..finally/except..end 语法关键字与内部例程 …… (219)
8.5 基本(except 型)异常处理 …… (221)
8.5.1 异常触发(Raise) …… (222)
8.5.2 多层(嵌套)的异常处理 …… (222)
8.5.3 异常展开(Unwind) …… (224)
8.5.4 异常响应(Notify) …… (227)
8.5.5 顶层异常处理 …… (229)
8.6 使用面向对象技术的异常类 …… (232)
8.6.1 异常列表、RaiseFrame 与 ExceptionRecord …… (233)
8.6.2 未知异常映射：ExceptObjProc …… (234)
8.6.3 不使用 SysUtils.pas 单元的应用程序 …… (236)

8.6.4 _RaiseExcept()、_RaiseAgain()与_DoneExcept()例程 …… (237)
8.6.5 异常对象 …… (240)
8.6.6 异常类型检测的依赖性问题 …… (247)

第 9 章 多线程的实现及调度 …… (251)

9.1 操作系统的进程与线程 …… (252)
9.1.1 线程的启动过程 …… (253)
9.1.2 主线程与用户线程启动过程的比较 …… (255)
9.1.3 线程有哪些独立元素 …… (256)
9.2 操作系统的线程调度概要 …… (257)
9.3 基本的多线程应用 …… (258)
9.4 线程变量(线程局部存储) …… (261)
9.4.1 线程局部存储(TLS)的实质 …… (262)
9.4.2 编译器的准备工作 …… (263)
9.4.3 静态 TLS：EXE 模块中的 TLS …… (265)
9.4.4 动态 TLS：DLL 模块中的 TLS …… (266)
9.5 再论 FS 段 …… (268)
9.6 线程类(TThread) …… (272)
9.6.1 线程执行体(Execute) …… (275)
9.6.2 线程与主线程的同步(Synchronize) …… (277)
9.6.3 控制台应用的多线程同步(第三方的 WakeMainThread 例程) …… (287)
9.6.4 线程中止和结束(Terminate) …… (292)
9.7 线程同步对象 …… (297)
9.7.1 线程同步方法 …… (298)
9.7.2 Delphi 中的线程同步类 …… (305)
9.7.3 线程对象的 WaitFor()方法 …… (310)
9.8 多线程环境下的内存管理器 …… (312)

第 10 章 资源的管理和使用 …… (315)

10.1 资源字符串 …… (315)
10.1.1 隐含的调用：LoadResString()例程 …… (315)
10.1.2 _InitResStrings()例程 …… (317)
10.1.3 _InitResStringImports()例程 …… (319)

10.1.4 _InitImports()例程 …… (320)
10.2 Delphi 编译的 PE 文件的默认资源 …… (322)
10.2.1 PACKAGEINFO 资源 …… (323)
10.2.2 PACKAGEOPTIONS 资源 …… (324)
10.2.3 DVCLAL 资源 …… (326)
10.2.4 DESCRIPTION 资源 …… (327)
10.3 其他 …… (327)
第 11 章 突破 Delphi 在应用开发上的限制 …… (329)
11.1 开发 DOS 保护模式程序 …… (329)
11.1.1 Delphi 1.0 与 DOS 16 位保护模式程序 …… (329)
11.1.2 WDOSX、Delphi 2.0-4.0 与 DOS 32 位保护模式程序 …… (331)
11.1.3 DWP、DWPL、Delphi 5.0-7.0 与 DOS 32 位保护模式程序中的图形接口 …… (333)
11.2 开发 Windows9x 上的驱动程序 …… (335)
11.2.1 Delphi 编译的.OBJ 文件 …… (335)
11.2.2 第三方的链接器处理 Delphi 的.OBJ 文件的问题 …… (336)
11.2.3 在 Delphi 中编写 VxD 驱动程序代码 …… (337)
11.2.4 链接目标文件并输出 VxD 驱动程序 …… (338)
11.3 开发 WinNT/2000 上的驱动程序 …… (339)
11.4 其他 …… (342)
术语表 …… (343)
参考书目 …… (347)

第一部分 I

Delphi 内核深入剖析（1）

DOS、Win3x 和 Win9x 的时代已经过去了，连 WinNT 的时代也已经远去。现在的应用程序开发已经不得不考虑在 WinNT 以上平台上的运行。因为即便是个人用户，其桌面也已经换成了 Windows 2000 Pro 或者 Windows XP。

然而作为程序员，我们无法忽视的事实是：Windows 系统在很多方面保留着自 Win 9x 以来的传统，在 API 这一层上，更是几乎完整地保留着自 Win9x 遗留下来的全部框架。

现在的应用程序已经要在.NET 平台上开发了。但是在.NET 之下，在开发框架之下，我们仍会触及到事件的本质：Win32 内核。

对于程序员来说，清楚 Delphi 在 32 位 Windows 下开发的一切细节，也许是灾难。因为那意味着与每一行代码相关联着的思考将会更多 所以有了封装，有了开发包，有了组件——仿佛一切都透明，都无须知道。

然而，如果一切都不清楚，那么，当灾难来临的时候，你将不知所措。因此，赶在灾难发生之前，还是静下来了解一下事件的本质吧。



章节指引

第 1 章虽然为读者引入了最小化内核 MiniDExe，但是，并没有详细地分析代码和运作机制。这些内容将留在第 5、6 章中讲述。

第 2、3 章是对一些深入内核时需要的基础知识的介绍，多数内容不直接涉及 Delphi 源码。

必须要注意的是第 4、5、6 章的内容，初读起来它们好像是倒置的。因为似乎更应该先讲述 PE 结构，再讲 Delphi 将内核挂在 OS 中的方法，最后再讨论单元和例程的初始化和结束化。但是，为了使读者尽快地触及到 Delphi 源码，我放弃了上面这种传统的叙述方法——钟表的构架如果没有问题，那么只须上一下发条，就可开始运转了——PE 结构就是那根发条。

第1章

最小化 Delphi 内核

Delphi 编译的.*EXE* 最小到底能有多少个字节？

很多人会用最简单的方式做一个“Hello World!”程序来给出答案。——在 Delphi7 中，会是 15K 字节。如果使用对话框，则马上会变成 382K 字节。

这里不去讨论为什么会有这种变化。因为我已经迫不及待地要向你介绍 Nico Bendlin 和他写的 MiniDExe 了。

——MiniDExe 的纪录是 3584 字节！

1.1 MiniDExe 如何实现内核最小化

Delphi 的单元隐含地引用了 *System.pas* 单元，那么 *System.pas* 单元自身如何被编译成.*DCU* 呢？查看*$(Source)\RTL\makefile* 文件可以看到，*System.pas* 是通过这样的命令行来被编译的：

```
$(DCC) sys\system -y -s $(RTLDEBUG) -n$(LIB)
```

这里最重要的就是隐含的编译选项：“-y”。它仅用于编译 *System.pas* 等被系统保留名字的内核单元，因此在 DCC32 的命令行帮助中是根本没有的。绝大多数资料中也没有介绍它。

MiniDExe 就是通过重写、重编译 *System.pas* 和 *SysInit.pas* 两个内核单元，全面摒弃了内核中的无关代码和多余变量、常量等，从而实现了最小化的。

1.1.1　MiniDExe 中的 System.pas 单元

```
unit System;

interface

procedure _InitExe;
procedure _HandleFinally;
procedure _halt0;

type
  TGUID = record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;

const
  Kernel32 = 'kernel32.dll';
  User32 = 'user32.dll';

var
  HKernel32: LongWord;
  HUser32: LongWord;

type
  TFNExitProcess = procedure(uExitCode: LongWord); stdcall;

var
  ExitCode: LongWord;
  ExitProcess: TFNExitProcess;

function LoadLibraryA(lpLibFileName: PAnsiChar): LongWord; stdcall;
function LoadLibraryW(lpLibFileName: PWideChar): LongWord; stdcall;
function LoadLibrary(lpLibFileName: PChar): LongWord; stdcall;
function GetProcAddress(hModule: LongWord;
  lpProcName: PChar): Pointer; stdcall;

implementation
// ...

end.
```

Delphi 源代码分析

这里只剩下了三个声明的过程：

```
procedure _InitExe;
procedure _HandleFinally;
procedure _Halt0;
```

其中_InitExe()是编译一个可执行文件时需要嵌入的初始化代码，_Halt0()则是可执行文件退出时的出口代码。显然，这是操作系统调入一个可执行文件必须的入口和出口代码。

_HandleFinally()则是编译 *SysInit.pas* 时所必需的，它用于挂接异常处理过程。

编译器还须要在 *System.pas* 中查找与 COM 接口相关的数据类型 TGUID。这使得可以在不改写编译器的情况下与当前的 COM 版本保持一致。

事实上，有了上述声明的部分，*System.pas* 已经可以用于编译 *.EXE* 可执行文件了。但是，为了让它能够做一点点工作——显示一个对话框——Nico 还必须继续为 *System.pas* 添上一些东西。

接下来定义了全局变量 HKernel32 与 HUser32，用于保存 OS 为当前进程映射的 *Kernel32.dll* 和 *User32.dll* 模块的句柄。这两个变量在_InitExec()例程中填入实际的值。全局常量 Kernel32 和 User32 仅用于定义上述两个模块的名字。

最后声明的是四个外部例程：

```
function LoadLibraryA(lpLibFileName: PAnsiChar): LongWord; stdcall;
function LoadLibraryW(lpLibFileName: PWideChar): LongWord; stdcall;
function LoadLibrary(lpLibFileName: PChar): LongWord; stdcall;
function GetProcAddress(hModule: LongWord;
  lpProcName: PChar): Pointer; stdcall;
```

它们都是在 Kernel32 模块中由操作系统实现的。如果用户的执行程序根本无须调用 Kernel32 与 User32 这两个模块中的任何例程，则上面的这些声明全都是可以省略的。

1.1.2 MiniDExe 中的 SysInit.pas 单元

```
unit SysInit;

interface

var
  TlsIndex: LongWord;
```

```
implementation

end.
```

这个单元中只剩下 TlsIndex 这个全局变量了，它用于保存一个线程 TLS 时隙的索引值。如果模块不定义线程局部变量，则 TlsIndex 不是必需的——尽管会导致编译器的错误警告。

1.1.3　MiniDExe 中的项目文件 MiniDExe.dpr

```
program MiniDExe;

type
  TFNMessageBox = function(hWnd: LongWord; lpText, lpCaption: PChar;
    uType: LongWord): Integer; stdcall;

const
  MB_ICONINFORMATION = $00000040;

begin
  TFNMessageBox(GetProcAddress(HUser32, 'MessageBoxA'))(0,
    'Written in pure Delphi!', 'Hello World!', MB_ICONINFORMATION);
end.
```

这个项目异常简单，是一个简单的 Win32API 的声明和调用。这样做是为了避免在 *System.pas* 中加入 Writeln()这样的输出例程代码。

1.2　一些其他的内核优化[①]

Delphi 提供了全部的内核源码和 VCL 源码，这为一些个人或第三方组织优化它们提供了方便，其中最有名的两个组织是 KOL 和“High Performance Code(optimalcode.com)”。一些常见的第三方的内核优化代码有：

- 内部例程优化：optimalcode.com 的 FastMath，Andrew N. Driazgov (andrewdr@newmail.ru)的 QStrings，以及 droopyeyes.com 的 FastString。

① 这些优化代码或者项目在本书中不会被具体讨论到。在这里提及它们，只是给读者分析内核时提供一些可供参考的资料。

- 内存管理器优化：在 QStrings 库中所包含的 QMem，optimalcode.com 发布的 HPMM，以及用于替换 ShareMem 的单元 FastShareMem 与 ShareMemRep。
- 异常管理器优化：KOL 发布的异常处理单元 *err.pas*。
- 系统单元的整体优化：KOL 发布的 System units replacement for D4-D7 和 Assarbad's system unit replace。

KOL 目前还在维护着各个 Delphi 版本的内核单元替换，并且已经对 *SysUtils.pas*、*Classes.pas*、*Variants.pas* 和 *Math.pas* 等单元进行了优化。而 optimalcode.com 站点如今已经关闭了。

也有一些非常优秀的、以个人名义维护的代码优化项目，例如 Assarbad 就发布过一个基于 Delphi4 优化的内核单元(但是更确切地说，Assarbad 并不注重对 System 单元的优化，他所关注的其实只是简化)。由 Dennis Christensen 负责的项目 Fast Code Project 也非常有名，目前的网址是 http://dennishomepage.gugs-cats.dk/FastCodeProject.htm，它甚至替代了 optimalcode.com 的位置。

此外还有一些散见于网络的内核例程的优化代码。例如这样的一个例程：

```
// same of function Math.Sing()
function Sign(n: Integer): Integer;
// returns -1 for n < 0
//          0 for n = 0
//          1 for n > 0
asm
  or eax,eax
  lahf
  movzx eax,ah
  shr eax,6
  sub eax,1
  neg eax
end;
```

1.3 为什么要研究最小化内核

回到 MiniDExe。

看起来没有必要把 *System.pas* 简化到如此之小——因为类似于 MiniDExe 中那样简化的内核几乎什么也做不了。但是，如果要从源代码一级分析 Delphi 是如何将对象、组件库、接口等框架技术包裹在应用程序之上的，从一个最小化的内核开始分析，是最方便、最清晰不过的了。

System.pas 展示了 Delphi 如何将自己的应用挂在操作系统中。这一点也很重要。因为，以此为原点，将更易窥见 Delphi 是如何实现多线程、内存管理、文件系统以及其他各种数据结构的。

自下面的章节开始，我将在这个内核之上，逐层包装代码，从而将 Delphi 内核的每一个实现细节具体展现在您的面前。

第 2 章

基本数据类型的实现

深入了解 Delphi 中各种数据类型的实现，是分析 Delphi 源代码的第一步。其中，有两点是必须清楚的：

- 数据类型的内存占用情况以及内存布局
- 强制类型转换

在本章中，主要讨论上面的两个问题。大多数情况下、Delphi 对于内部数据结构的处理代码，都放在编译器和内核例程中。因此，本章将使用相对较多的汇编代码，来解释 Delphi 对各种数据类型的处理。

常量不是本章讨论的重点。

有关基本数据类型更详细的讨论，读者可以参见参考书目：《PASCAL 精要》、《DELPHI 技术手册》和《Object Pascal 参考手册》。

2.1 基本数据类型

从 Pascal 到 Delphi，语言中最基本的数据类型如表 2-1 中所列。

表 2-1　各种语言的基本数据类型

分　类	数据类型	长度[①]	相关类型(部分)	备　注
简单类型	整数类型(integer)	1-8	Integer, Cardinal Shortint, Smallint Longint, Longword Byte, Word, Int64	除Int64之外，所有的整数类型在内存中的存放都非常简单。由于使用标准化的存储方法，整数类型可以和其他语言方便地进行数据交互
	字符类型(character)	1-2	AnsiChar, WideChar	标准化存储，一个字符与一个字节并非同一概念
	布尔类型(Boolean)	1-4	Boolean, ByteBool WordBool, LongBool	标准Pascal中，Boolean是以一个字节存放的。其他长度的布尔类型，通常只在多语言交互时使用
	枚举类型(enumerated)	1-4		参见编译指示字$MINENUMSIZE或$Z
	子界类型(Subrange)			
	实数类型(real)	4-10	Real48, Single, Currency, Double, Extended, Comp	六字节的Real48类型在Delphi较早的版本中叫做Real。该类型是为了保持向后兼容而存在的
字符串	字符串类型(String)	1-256/4	ShortString AnsiString WideString	对于短字符串，长度为1~256字节。对于其他字符串类型，长度为4字节
构造类型	集合类型(set)	1-32		集合的元素最多为256个，以位数组的格式存放
	数组类型(array)	4		静态数组与动态数组的数据结构实现完全不同。动态数组为4字节，静态数组的长度取决于具体定义
	记录类型(record)			取决于具体定义
	文件类型(file)	332/460		文件变量以332字节的记录存在。文本文件变量的记录占460字节
指针类型	指针类型(pointer)	4	PInteger, PPointer	
过程类型	过程类型(procedure)	4		

① 这里的“长度”仅指参考长度(字节)，在代码中可以用 SizeOf() 来取得该值。一些较复杂的数据类型，以及几乎所有的用户定义类型的长度，都在表中所列出的“长度”值之外。同时，这里的“长度”值还表明：如果该数据类型的变量被声明，则系统“至少”为该变量初始化的字节数。

2.2 变量与常量

2.2.1 全局变量与局部变量

根据变量在源代码中定义的位置不同，可以分成局部变量和全局变量两种。此外，在多线程应用程序中，可以定义线程变量(ThreadVar)。下面的例子描述局部变量和全局变量在内存占用上的不同：

```
{$APPTYPE CONSOLE}
uses
  SysUtils;

var
  GVar1 : Integer;
  GVar2 : Integer;

function GetAddr(Const aVar) : String;
begin
  Result := IntToHex(Integer(@aVar), 8);
end;

procedure WriteLocalVar;
var
  LVar1 : Integer;
  LVar2 : Integer;
begin
  Writeln('Local Var 1 : ', GetAddr(LVar1));
  Writeln('Local Var 2 : ', GetAddr(LVar2));
end;

begin
  Writeln('Globl Var 1 : ', GetAddr(GVar1));
  Writeln('Globl Var 2 : ', GetAddr(GVar2));
  WriteLocalVar;

  readln;
end.
```

输出结果是类似于这样的：

```
Globl Var 1 : 0040A798
Globl Var 2 : 0040A79C
Local Var 1 : 0012FF90
Local Var 2 : 0012FF8C
```

全局变量在应用程序的数据区分配，而局部变量在应用程序的栈上进行分配。因此，相对于定义的顺序，多个全局变量的地址是递增的，而局部变量递减。此外，由于每次调用函数时，栈顶可能发生变化，因此，局部变量的地址是变化的，而全局变量是不变的。

全局变量在编译期就被决定，它存在于可执行模块(*.EXE* 或 *.DLL* 等)的文件影像内部。因此，在载入文件的同时，全局变量的内存就被分配了[①]。而局部变量是在例程被调用的时候才被分配的。

局部变量总是在栈上分配。

2.2.2　动态分配的内存

变量在运行期动态分配的内存是一个例外。这里说的“动态分配的内存”[②]是指用 GetMem()等在代码中显式分配的内存。例如系统源码中对对象、字符串(短字符串除外)、动态数组[③]等进行的内存分配，以及在用户代码中使用 GetMem()来分配的内存等。这些内存将在堆上进行分配。

堆分配与栈分配不同。但通常不会有太多人注意到这个细节。下面的例子阐述了这一点。

```
{$APPTYPE CONSOLE}
uses
  SysUtils;

function GetAddr(Const aVar) : String;
begin
  Result := IntToHex(Integer(@aVar), 8);
end;

function GetDataAddr(Const aVar) : String;
begin
```

① 常量与全局变量的分配是不同的，但是，类型化常量与全局变量在同一内存空间中被分配，且使用相同的分配规则。

② 官方资料中，“动态分配的内存”被定义为“动态变量”，但由于它实际上是变量的数据体部分，因此，为了将它与普通变量区分开来，这里称为变量“动态分配的内存”。一些其他的函数也可以完成这样的内存分配，例如 ReallocMem()、Initialize()、StrAlloc()和 StrNew()等。

③ 静态数组也将直接在栈上进行分配。这意味着在局部变量中声明过大的静态数组是不安全的。细节请参见 2.3.3 小节中的“数组类型”。

```
  Result := IntToHex(Integer(aVar), 8);
end;

procedure WriteLocalVar;
var
  StackVar : String;
begin
  StackVar := '0123456789';
  Writeln('Stack Var    : ', GetAddr(StackVar), 'Length: ':15,
    SizeOf(StackVar));
  Writeln('Variable Str : ', GetDataAddr(StackVar), 'Length: ':15,
    Length(StackVar));
end;

begin
  WriteLocalVar;

  readln;
end.
```

输出结果是类似这样的：

```
Stack Var      : 0012FFA0      Length: 4
Variable Str   : 00BF0960      Length: 10
```

在这个例子中。局部变量 Stack Var 只在栈上分配了 4 Bytes。但是作为字符串，它的 10 Bytes 的内容都被分配在堆中。因此，输出结果中的“Stack Var”与“Variable Str”的地址并不一样。

2.2.3 换一个方式来理解

通常可以认为：变量由两个部分组成，即变量自身和变量的内存占用。

“变量自身”遵循变量分配的基本原则，其大小可以参见表 1-1，也可以直接由函数 SizeOf()取出。如果是简单类型，则存放变量值。

如果变量是复杂的类型(如一些构造类型或字符串类型)，则“变量自身”仅存放“变量的内存占用”的指针。“变量的内存占用”的大小取决于该变量的数据结构，通常有专用的或独立设计的函数来取其大小。例如函数 GetLength()用来取字符串或者数组大小。

如果没有特殊的代码实现，“局部变量”存在于栈，而“变量的内存占用”存在于堆。[1]

基于上一小节的例子，图 2.1 简单地描述了这种关系[2]。

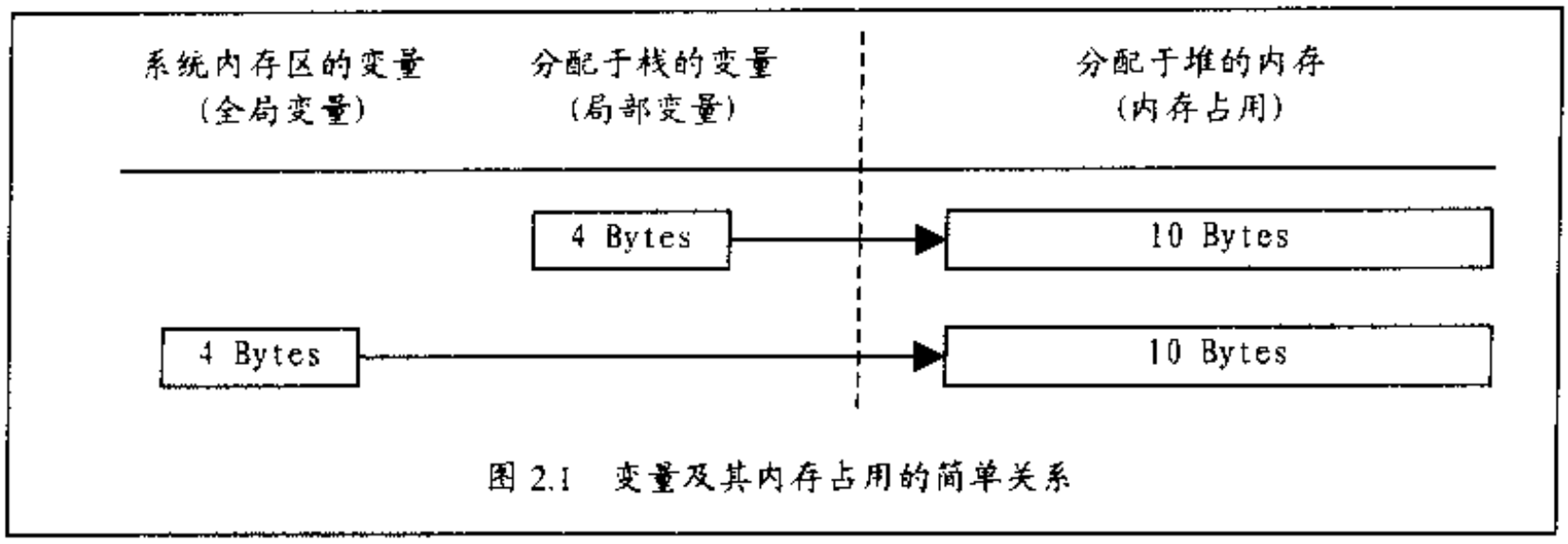


图 2.1　变量及其内存占用的简单关系

全局变量和常量的内存分配的更多细节，可以参见第 6 章“Delphi 的积木艺术(PE)”中第 3 节有关“数据节与代码节”的内容。

2.2.4　常量

关于常量，上面已经提到：类型化常量与全局变量在同一内存空间中被分配，且使用相同的分配规则。——但这不包括有关“变量的内存占用”的规则。

此外，普通的、非类型化的常量(真常量)与变量也存在着一些不同。

基于 Delphi 的常量定义规则[3]，编译器将常量作为立即数处理：将值直接编码到指令中。因此它们不可能有内存占用，当然也不可能有具体的内存地址值。

字符串常量是例外的。显然字符串不能当作立即数处理，因此编译器总是将字符串常量与代码放在一起，并将它的地址指针直接编码到指令中。

① 以后的行文中，变量的这两个部分将分别以“变量”和“变量的内存占用”来称呼。需要一再强调的是：变量的内存占用、动态变量和变量“动态分配的内存”是同一概念。

② 这种情况下，变量的 4 字节总是可以通常类型强制，当作指向其内存占用的指针进行操作。

③ 除了字符串和集合，Delphi 中定义的常量大小都不能超过 8 字节(Int64)，否则必须声明成“类型化的”，而这种情况下，Delphi 按照变量的规则进行内存分配。

下面的代码揭示字符串常量与变量在内存区使用的不同之处：

```
{$APPTYPE CONSOLE}
{$DEFINE MORECONST}
uses
  SysUtils;

const
  DefineCon1 = '0123456789';
{$IFDEF MORECONST}
  DefineCon2 = '9876543210';
{$ENDIF}

var
  DefineVar1 : string = '0123456789';

function GetAddr(Const aVar) : String;
begin
  Result := IntToHex(Integer(@aVar), 8);
end;

function GetDataAddr(Const aVar) : String;
begin
  Result := IntToHex(Integer(aVar), 8);
end;

begin
  Writeln('Const 1 Addr  : ', GetAddr(DefineCon1), 'Length: ':15,
    SizeOf(DefineCon1));
  Writeln('Const 1 Data  : ', GetAddr(DefineCon1[1]), 'Length: ':15,
    Length(DefineCon1));
{$IFDEF MORECONST}
  Writeln('Const 2 Addr  : ', GetAddr(DefineCon2), 'Length: ':15,
    SizeOf(DefineCon2));
{$ENDIF}

  Writeln;
  Writeln('Variable Addr : ', GetAddr(DefineVar1), 'Length: ':15,
    SizeOf(DefineVar1));
  Writeln('Variable Data : ', GetDataAddr(DefineVar1), 'Length: ':15,
    Length(DefineVar1));

  readln;
end.
```

尝试定义和取消编译条件“MORECONST”并编译运行，可见输出的结果都是一致的。因此是否定义常量 DefineCon2，对于变量 DefineVar1 的起始地址没有影响。这意味着字符串常量与变量使用的是不同的内存空间。在上例中，DefineCon1 的地址与其数据在同一地址，这是由于字符串常量总是直接传入实际地址。

字符串常量总是被分配内存。关于其分配内存的细节，可以参考第 6 章中“数据节与代码节”的内容。

元素数为 64~256 个(即大小超过 4)的集合常量，也使用与字符串常量类同的规则。

使用类型强制转换方式声明的常数，按照转换到的目标类型来分配内存。但是，它不是类型化常量。例如：

```
const
  // 普通常量，按约定，占用空间取决于能容纳该值的最小类型：Byte
  I_Byte = 5;

  // 类型强制转换的常量，占用空间取决于转换到的目标类型：DWORD
  I_DWord = DWORD(5);

  // 类型化常量，与 I_DWord 占用空间大小相同，但却是在数据节中。
  I_TypeDWord : DWORD = 5;
```

2.3　数据结构的实现

2.3.1　简单类型

简单类型的变量直接分配内存。可以通过上面的代码非常方便地取得它在内存中的地址及所占用内存的长度。

短字符串和集合类型，也使用与简单类型一致的规则。

简单类型的变量在内存中具有特定的格式，比如实数、枚举类型。有关其具体的存储格式，可以查阅参考书目中的《DELPHI 技术手册》以及 IEEE 标准文档。

2.3.2　字符串

Delphi 中的字符串格式多而且混乱。

“多”是因为 Delphi 一方面要继承自 Pascal 以来的语言习惯，另一方面又要与 Windows 制定的内部数据结构和外部编程接口相兼容。“混乱”源自 Delphi 总是试图让开发人员可以直接在各种字符串之间任意交换数据，而无须了解各种字符串的内部格式——因而，这种混乱的感觉为企图深入了解每一行 Delphi 语句的奥秘的程序员所深受，而对于只用它来完成一个应用的开发者来说，一切都是透明的。

按照官方的资料，字符串只有短字符串、长字符串和宽字符串三种。但下面将能与它们直接赋值的“Null 结束字符串”和“字符数组”也一并讲述。

■ 短字符串

短字符串至少分配一个字节(即便是“空”串)。这个字节的地址亦即字符串在内存中占用的首地址，用于存放字符串长度，称为计数位。自其后，开始存放字符串中的各个字符。因此，短字符串可以存放 0~255 个字符。

短字符串可以有两种声明方式：

```
Var
  var1 : String[20];
  var2 : ShortString;
```

var1 的声明限定字符串长度为 0~20 字节；var2 的声明使用默认值，因而限定为 0~255 字节。

Delphi 总是按最大上限为短字符串分配内存。这样，最少的情况下，var1 将占用 20+1 字节，而 var2 将占用 255+1 字节——这 1 字节是为 var1[0]和 var2[0]的计数位保留的。

由于堆栈以 4 字节为单位进行分配[1]，因此，var1 在堆栈中将被分配 24 字节，而 var2 将被分配 256 字节。源于堆栈的这种分配，在应用程序内存区中，短字符串类型化常量采用与堆栈分配一致的长度分配。

短字符串格式是从 Pascal 继承而来的。在 Pascal 中一个字符串与字符数组是数据兼容的。这表明可以直接在字符数组与字符串之间赋值。

① 编译指示字$Align 只对记录、类和对象有意义。它不影响字符串在堆栈和应用程序内存区中的内存分配规则。

■ 字符数组

可以用声明数组的规则声明字符数组：

```
var
  var1 : array [0..20] of char;
```

在内存长度上，字符数组使用与短字符串一致的内存分配规则。因此，数组 var1 将占用 24 字节内存。

字符数组同时遵守数组在内存使用上的规则。

■ Null 结尾字符串

PChar 类型是典型的使用 Null 结尾字符串规则的数据类型。

此外，也可以将任意的内存块视作 Null 结尾字符串来定义和操作。例如用 GetMem()获得的一块内存。——几乎所有的内存块都满足这样的规则。因此，Null 结尾字符串是一个弱类型检测的定义。在 WinAPI 中有大量的地方使用了 Null 结尾字符串，所以，很多地方随便传个指针，也可能导致无法预测的结果。

很多时候，使用 PChar 来定义一个普通的指针，其目的并不是要将指针处理成字符串，而是为了地址运算的方便。因为 PChar 是编译器支持的数据类型，编译器在必要的情况下重定义 Char 类型，就可以同时支持纯 WideChar 或 AnsiChar 核心的操作系统，例如纯 Unicode 的操作环境。这比定义成 PByte 类型要更安全。

■ 长字符串(AnsiString)

缺省以 String 关键字定义的，或者直接以 AnsiString 关键字定义的字符串变量都是长字符串。长字符串兼有短字符串和 Null 结尾字符串的特征，同时能够以字符数组的方式进行操作。

长字符串变量总是占用 4 个字节，所以绝大多数时候，可以把长字符串当成指针来操作。——在 Delphi 内部更是这样。在 TStrings 类中，几乎所有的操作都是针对于这 4 个字节的。这使得 TStrings 拥有与 TList 相当的速度和性能。

长字符串的内存占用相对来说要复杂些：

- 空值的长字符串的内存占用为 0 字节
- 非空值的长字符串的内存占用为：串长+8 字节+1 字节

实际上 Delphi 使用一个空指针来表示空值的长字符串。所以空字符串在堆中没有内存占用。

非空值的情况下，除了为字符串分配应有的内存空间之外，Delphi 还在负偏移的位置占用了 8 个字节，分别保存了串长和串引用计数，其内存布局如图 2.2 所示。

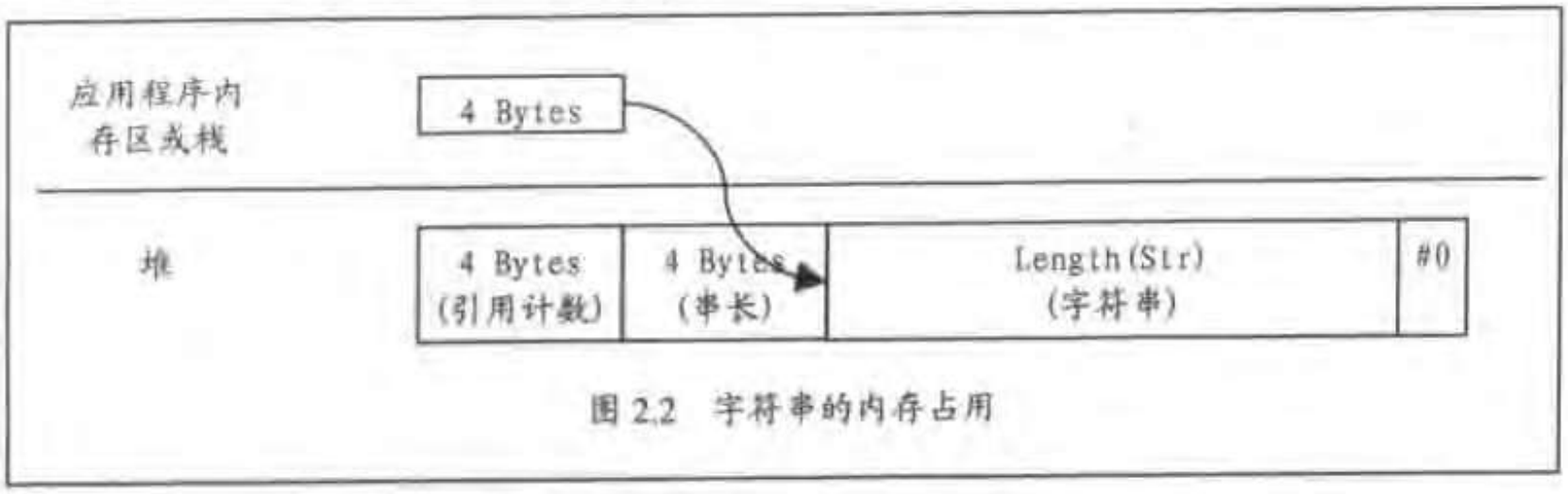


图 2.2 字符串的内存占用

下面的代码用于直接操作引用计数和串长：

```
function GetStringRef(Const Str:AnsiString) : Integer;
begin
  if Str = '' then
    Result := -1
  else
    Result := PInteger(Integer(Str)-8)^;
end;

function GetStringLen(Const Str:AnsiString) : Integer;
begin
  if Str = '' then
    Result := 0
  else
    Result := PInteger(Integer(Str)-4)^;
end;

procedure SetStringRef(Const Str:AnsiString; Ref:Integer);
begin
  if Str <> '' then
    PInteger(Integer(Str)-8)^ := Ref;
end;

procedure SetStringLen(Const Str:AnsiString; Len:Integer);
begin
  if Str <> '' then
    PInteger(Integer(Str)-4)^ := Len;
```

```
end;
```

直接修改引用计数是不安全的，系统对字符串有效性的管理将会因此紊乱。而直接修改字符串的长度，可能比操作 SetStringRef()更为危险。SetStringLen()会导致系统对字符串的内存管理紊乱。这主要有两方面：

- 结束符#0 不会自动添加，结果串在作为“Null 结尾字符串”处理时，不会有预想的改变。
- 变量占用内存并没有实际发生变化，访问增加长度后的字符串可能会导致内存存取错误。

因此，除非能确知这样做的后果，否则不要使用函数 SetStringRef() 和 SetStringLen ()。

■ 字符串的引用计数与写复制

由于有引用计数的关系，所以，Delphi 会管理两个相同的字符串，以及由赋值而得的字符串。这使得多个字符串的内存占用分析变得复杂起来(例如需要先比较两个字符串是否是同一地址的引用)：

- 如果在编译期，能确知两个字符串相同，则编译器使两个字符串使用相同的内存占用。
- 赋值操作会使得两个字符串使用相同的内存占用。
- 修改字符串的操作，会导致“写复制(copy-on-write semantics)”，并且使相同内存占用的两个字符串，开始使用不同的内存占用。
- 直接访问字符串中的元素(以 s[i]的形式访问一个字符)，会导致整个字符串写复制。

图 2.3 展示了这种关系。

由于短字符串、Null 结尾字符串和字符数组没有引用计数和与长字符串相同的专用计数位，所以，上面的关系在这些类型的字符串中并不完全存在。

■ 宽字符串

WideChar 是一种 16 位 Unicode，它占用两个字节内存。基于 WideChar 的宽字符串(WideString)与 AnsiString 的内存结构不完全相同。但是，和大多数人预想的不同：WideString 并非正好比 AnsiString 多占用一倍的内存空间。例如：

```
var
  Str1 : String = 'abc中文';
  Str2 : WideString = 'abc中文';
```

在这个例子中，Str1 占 3+4=7 字节的空间，而 Str2 占用 6+4=10 字节；只有字符串“abc”占用的长度发生了变化。

Delphi 6 的《语言手册》中关于宽字符串的一段描述是有错误的，手册中指出“s[i]代表 S 中的第 i 个字节(而不一定是第 i 个字符)”。实际却正好相反，在宽字符串中，s[i]代表第 i 个字符，却不一定是第 i 个字节。

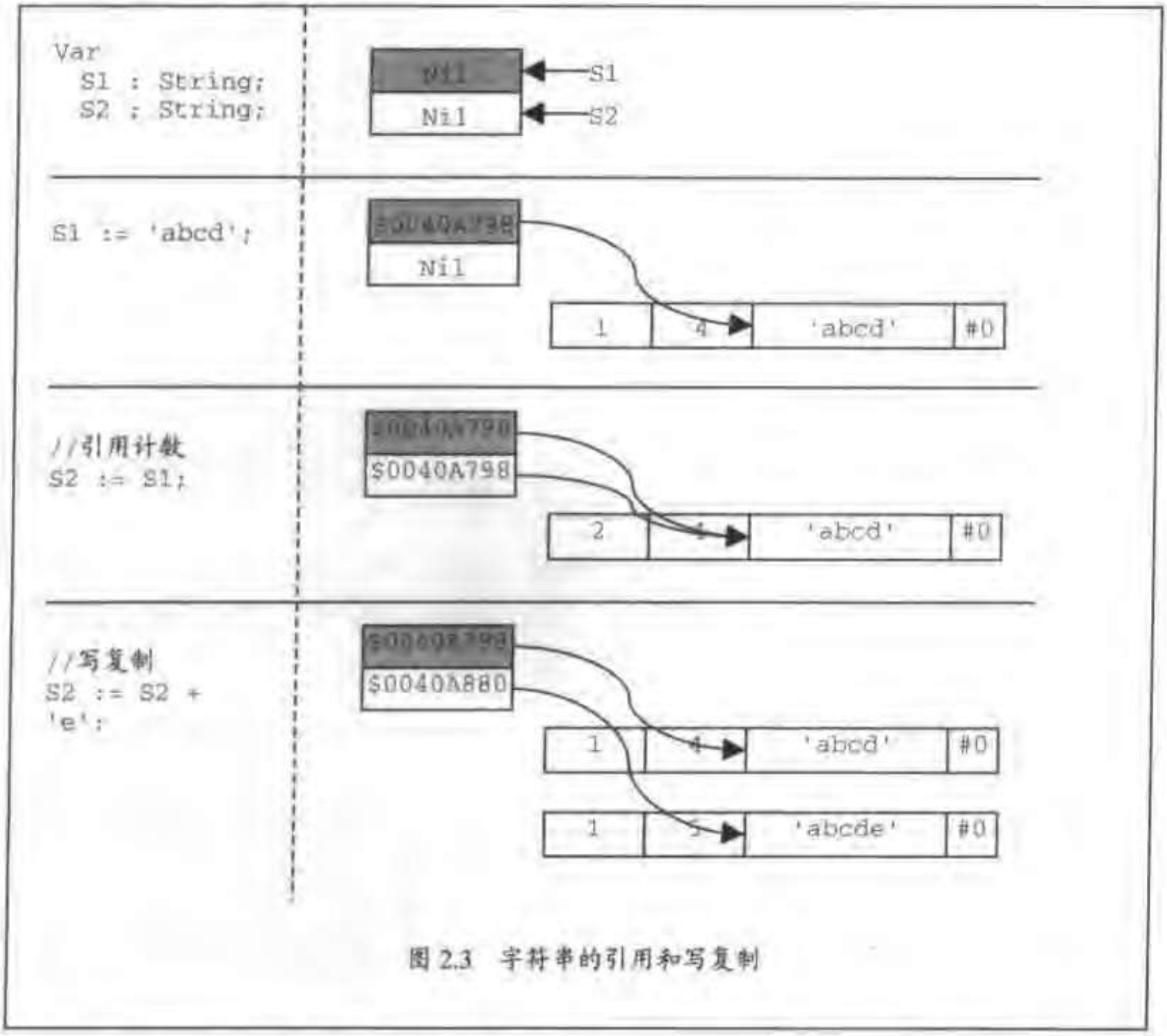


图 2.3 字符串的引用和写复制

Delphi 中使用基于 AnsiChar 的数组来操作 WideString 是危险的。这种情况下应该使用基于 WideChar 的数组。为了使与标准 Pascal 相同语义的代码在 Delphi 中完全正常地被编译，Delphi 将 s[i]解释成一个字符，而无论它是 AnsiChar，还是 WideChar——Char 与

Byte 的对应关系被打破之后，很多麻烦都出来了，相信深入过中文操作系统的人都清楚这一点。

在内存结构上，WideString 呈现出奇特的一面，举例来说：

```
Var
  Str : WideString = 'abc';
```

该定义在内存中的结构如图 2.4 所示。

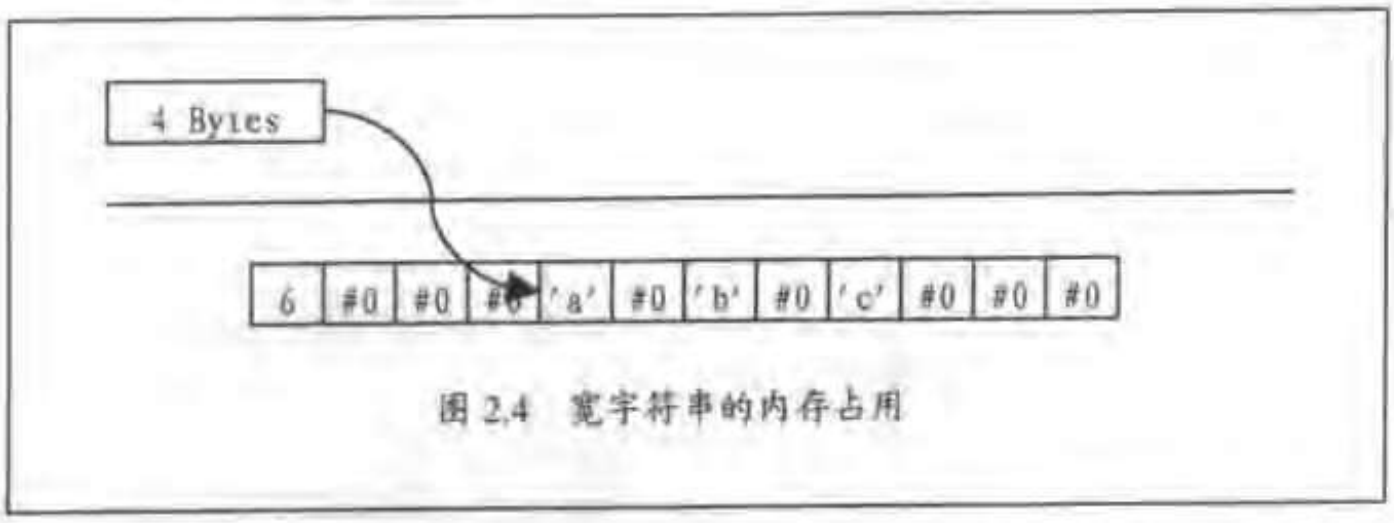


图 2.4　宽字符串的内存占用

为了和 Null 结尾字符串兼容，WideString 也在结尾添加了一个#0 字符(两个字节)。

为了和 AnsiString 兼容，WideString 在字符串的负偏移处也加上了长度位，占四字节。但是，WideString 没有引用计数，因而字符串引用计数的规则对 WideString 也不适用。

在长度位上，保存的值是字符串占用的字节数，而非字符数。所以，上例串长是“6”，而不是“3”。但是，如果用 Length()操作，却又会得到串长为 3。由此可以想到：直接存取长度位是不安全的。

下面的代码用 Pointer 类型作过渡，使编译器在语义上失去对 WideString 的理解。

```
Var
  Str : WideString = 'abc';
  // ...
  Writeln(Str);
  Writeln(Length(Str));                          //显示为3
  Writeln(Length(String(Pointer(Str))));         //显示为6
```

■ 字符串混用的代价：计数位与结束字符的维护，以及内存复制

字符串的相关类型很多，其中比较常见的是 PChar，它的定义其实是这样的：

```
Type PChar = ^Char;
```

然而作为内部数据类型，Delphi 对 PChar 的处理是很多的，其中主要包括的是在各种字符串之间的数据类型转换上。几乎每个与其他类型的字符串相关的 PChar 的操作中都涉及到长度计数和引用计数问题。而且为了与操作系统所理解的 PChar 规则一致，Delphi 还要为它的 PChar 维护一个结束字符 Null。

除了 Null，PChar 没有其他额外的域，这使得将一个长字符串转换成 PChar 是不需要任何额外代码支出的。但在开发者的具体代码中，系统内部的引用计数和写复制机制会使得整个关系变得让人捉摸不定。如果开发者需要了解每一行代码的细节，就应该回到前面的章节仔细阅读，并使用 GetStringRef()和 GetDataAddr()两个函数来跟踪具体的代码。

然而，在短字符串与 PChar 之间转换绝非易事。因为短字符串没有结束字符 Null ,所以不能直接在它与 PChar 之间转换，而必须先将 ShortString 转换到 AnsiString。而这将导致一个复制操作。下面的代码说明了这一点：

```
var
  s1: ShortString = 'abcd';
  s2: String = '1234';
  p : pChar;

  // ...
writeln(GetStringRef(s2));
writeln(GetDataAddr(s2));
writeln('---');

//赋值操作将不会导致引用计数的增加，p指向地址与s2的内存占用地址一致，表明没有
//复制操作发生
p := pchar(s2);
writeln(GetStringRef(s2));
writeln(GetDataAddr(p));
writeln('---');

//通过String()转换将导致写复制操作发生：s[1]的地址与p指向地址并不相同
// p := pChar(s1);     <-- 这样的语句是不会被编译通过的
p := pChar(String(s1));
writeln(GetAddr(s1[1]));
writeln(GetDataAddr(p));
```

更多的麻烦表现在 WideString 与 AnsiString 的转换上。这主要涉及到字符串长的重新修正与内存重新分配。由于没有引用计数，所以无论是否有类型转换操作，WideString 的赋值总是将导致新的内存占用。只不过 WideString 之间的赋值不会导致字符串长的重新修正，而将 WideString 类型的变量赋给 *AnsiString* 类型的变量时，必须重新计算字符串

长。

因此，使用 WideString 的系统的效率会低于 AnsiString。这一切都源自于没有引用计数机制。同时，混合使用 AnsiString 与 WideString 的系统的效率将会更差——因为系统必须为重新计算字符串长而付出额外的代价。

内存复制的情况表现在：

- AnsiString 与 ShortString 之间赋值和强制类型转换。
- WideString 与任何类型的字符串赋值和强制类型转换。
- 将字符串指针赋给字符串。

以下操作不会导致内存复制：

- 将字符串赋给指针(包括无类型指针和各种字符串指针)。
- 对上面操作的目标指针的任意操作，不会导致源字符串的内存复制或引用计数变化。

■ 由引用计数导致的问题

由于编译器通常只能理解直接赋值和类型转换中的字符串引用，并据此来修正引用计数。所以在直接内存操作的情况下，常常会使长字符串的引用计数失去维护。

一个极其典型的例子源自于对记录 Move()操作，如下：

```
type
  TRec = Record
    Index  : Integer;
    Str : String;
  end;
var
  R1, R2 : TRec;

// ...
  R1.Index := 1;
  R1.Str := 'Test';
  Writeln('Ref for R1.Str : ', GetStringRef(R1.Str));

//  R2 := R1;
  move(R1, R2, SizeOf(R2));
  Writeln('Ref for R1.Str : ', GetStringRef(R1.Str));
```

Delphi 源代码分析

```
Writeln('Addr : ', GetDataAddr(R1.Str), '=', GetDataAddr(R1.Str));

R2.Str := 'Now, Error!';
```

根据 Delphi 的语法，调用 move()的这一行代码意味着两个数据结构(记录)完全被复制。但这并不等于“R2.Str := R1.Str”这样的操作同时发生，因此，R1.Str 的引用计数并没有发生变化，而 R2.Str 却成了一个有效的字符串。

接下来的一行代码：

```
R2.Str := 'Now, Error!';
```

用于“破坏”引用计数。由于 R2.Str 是 Delphi 认可的有效字符串，所以在执行这行代码的同时，引用计数将减 1，从而变成 0。按照字符串的处理规则，引用计数为 0 的字符串将被释放内存(清除)。这时，R1.Str 指向了一个不能被访问的内存块。

图 2.5 进一步地描述这种现象。

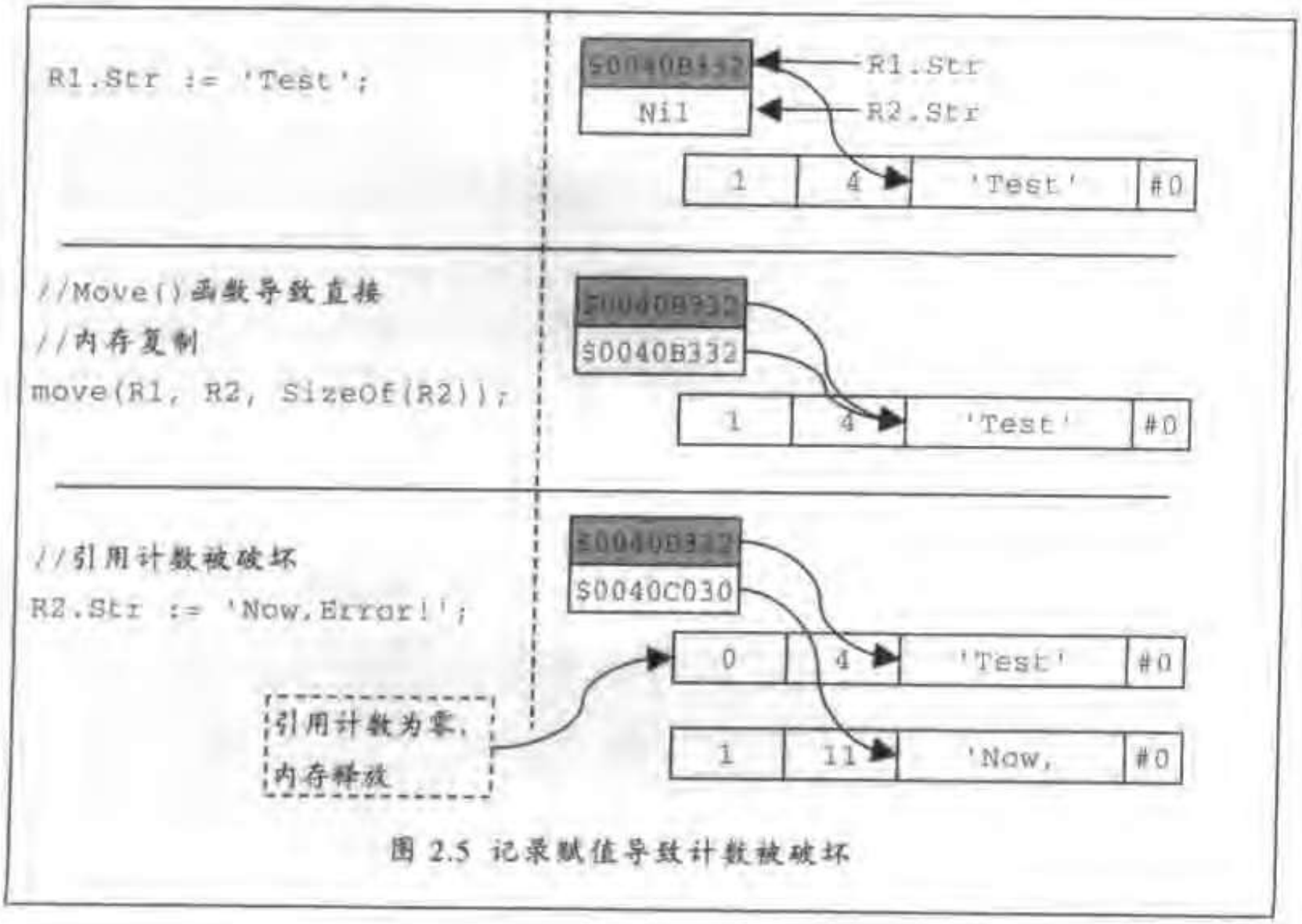


图 2.5 记录赋值导致计数被破坏

替代 move()的安全方法是直接使用记录赋值“R2 := R1;”。在编译期，Delphi 将会

为记录赋值操作加入修正引用计数的代码。

2.3.3　构造类型

■ 集合类型

集合类型与简单类型一样，没有动态分配的内存。集合的每个元素占一个位，因为 Delphi 最大支持 256 个元素的集合，所以使用 1~32 字节来存放各个位数据。

集合类型的局部变量总在栈上分配。

■ 数组类型

不论数组元素是什么类型，静态数组的局部变量总会在栈上分配。如果堆栈大小不够，将会导致异常。Delphi 在编译期不会计算栈的耗用。下例暗藏着一个危险的 BUG：

```
procedure StackOver;
var
  S : array [0..4100] of String[255];
begin
end;
```

Delphi 默认栈最大为$100000 字节，由于 String[255]占用 256 字节，且总是栈分配，则可以计算得知，即使应用程序不耗用任何栈空间，最多也只能分配$100000/$100=4096 元素。因此，上例中调用 StackOver()总会发生异常：Stack overflow。然而编译器并不能检测到这个错误。

可以使用$MaxStackSize 编译指令调整栈空间的大小。

与静态数组不同，动态数组变量只占用 4 个字节。下例简单地说明它们之间的不同：

```
procedure ArrayTest;
var
  S : array [0..1000] of String[255];
  S1: array of String[255];
begin
  SetLength(S1, 1001);

  Writeln(SizeOf(S));        // 输出：256256
  Writeln(SizeOf(S1));       // 输出：4
end;
```

动态数组在堆中动态分配内存，并且采用与长字符串完全一致的内存结构：在负偏移处保存着 8 字节的引用计数和长度计数。SetLength()隐含地调用了 ReallocMem ()。

■ 记录类型

缺省情况下，记录会以对齐方式存储。在编译期，Delphi 用一套复杂的方法来计算对齐格式记录的内存占用情况。Packed 关键字和{$Align}编译指令会对 Delphi 的这种行为造成影响。

记录总是直接分配内存。因此，与静态数组一样，在例程中定义过大的记录类型的局部变量时，也可能导致栈溢出——如果需要定义复杂到$100000(或更多)字节的记录的话。

变体记录(有可变部分的记录类型)使用能够容纳可变部分最大长度的空间来存储。Delphi 约定变体部分必须出现在记录定义的最末部分，这使得在计算和分配空间时，可以通过直接在记录后面附加空间的方式来实现。如果要在记录的中间部分声明变体，则须单独声明一个变体记录类型，例如：

```
TIIDUnion = record
  case Integer of
    0: (Characteristics: DWORD);
    1: (OriginalFirstThunk: DWORD);
end;

TImageImportDescriptor = record
  Union: TIIDUnion;
  TimeDateStamp: DWORD;
  ForwarderChain: DWORD;
  Name: DWORD;
  FirstThunk: DWORD;
end;
```

变体记录使得编译器可以将一个内存空间解释为多种类型，这意味着它可以用于隐含地实现数据类型转换。

■ 文件类型

文件类型是基于压缩(Packed)的记录类型来实现的。类型文件和无类型文件都采用同一记录类型，在*$(RTL)\System.pas* 中，可以查阅到它的定义：TFileRec。文本文件采用专用的记录类型，在*$(RTL)\System.pas* 中定义为 TTextRec。

文件类型的内存分配规则可以参见记录类型。

文件类型的详细实现和内部例程在第5章的“32位的DOS：控制台应用程序”中讲述。

2.3.4　指针类型

指针总是占用4字节的内存。任何时候都可以将指针作为整型使用，并用整型运算做地址运算。有类型指针可以使用内部例程Inc()和Dec()进行地址运算。

2.3.5　过程类型

过程变量只占用4字节的内存，它实际上是以指针形式存在的。过程变量只存储例程代码地址的入口地址，真实的例程代码只存在于应用程序的代码段中。尽管如此，Delphi采用一种与指针不同的语义来理解过程——因此，不能在指针与过程变量之间做强制转换。下例说明过程是采用指针的形式存储的：

```
procedure TestProc;
type
  TRec = packed record
    Int : Integer;
    Proc : Procedure;
  end;
var
  Rec : TRec;
  PInt : PInteger;
begin
  Rec.Int := $1234FFFE;
  Rec.Proc := TestProc;
  PInt := @Rec;
  Writeln('Rec.Int  : ', IntToHex(PInt^, 8));

  //不能直接做强制转换，下一行代码出错
  //PInt := PInteger(Rec.Proc);
  Inc(PInt);
  Writeln('Rec.Proc : ', IntToHex(PInt^, 8));
end;
```

应用程序代码段既不是栈访问，也不是堆访问。所以，过程类型较其他变量类型更为特殊，通常情况下，没有办法分配过程类型变量的“动态内存”。

应用程序的代码通常是在编译期由编译程序生成的，并且在程序文件内部组织完成。这样

一旦编译过程结束，例程代码也就确定了。在应用程序调入时，系统会将包含这些例程代码的文件映像部分定义为代码段，代码段是只读的。

因此，在代码中，可以简单地将过程赋给一个过程变量，但如果要真的将过程变量作为指针用来修改目标内存的数据，就会出错。修改上例中的代码来示范这种现象：

```
procedure TestFixProc;
type
  TRec = packed record
    Proc : Procedure;
  end;
var
  Rec : TRec;
  PInt : PInteger;
  PPtr : PPointer;
begin
  Rec.Proc := TestFixProc;
  PPtr := @Rec;
  PInt := PPtr^;

  //直接修改代码段导致出错
  try
    PInt^ := 1;
  except
    on e:Exception do
      Writeln(e.message);
  end;
end;
```

2.4 数据结构相关的例程

编译器实现了基础数据类型(以及相关的强制转换机制)，最小化的 Delphi 内核实现了可执行模块装载程序。因此，利用这样的最小化内核能够编译的应用只能做到“数据结构＋内置运算符”。

在 Delphi 中使用数据结构，并不是要编译器简单地理解这些数据结构。更重要的是，必须有一套相关的内核函数和过程的支持。在源代码中，这些与基本数据结构相关的例程名通常是以“_”开始的，它们虽然声明在 Interface 节中，却无法在用户代码中直接调用。这些例程包括：

- 标准 Pascal 的内置例程；
- 短字符串操作例程；

- 长字符串操作例程；
- 宽字符串操作例程；
- 未分类的字符串操作例程；
- 集合操作例程；
- 数组(含动态数组)操作例程；
- 记录操作例程。

2.4.1　标准 Pascal 的内置例程

```
procedure _COS;
procedure _EXP;
procedure _INT;
procedure _SIN;
procedure _FRAC;
procedure _ROUND;
procedure _TRUNC;
procedure _Pow10;
```

Delphi 通过内嵌汇编，用浮点运算指令完成了上述例程。在这部分的(包括其后的)代码中，有一个有趣的注释：

```
{ Procedures and functions that need compiler magic }
```

应该注意到，COS()的接口定义与 Delphi 帮助中并不相同。在帮助中，定义是：

```
function Cos(X: Extended): Extended;
```

这样的差异也表现在其他的例程中。因此，“compiler magic”表明了 Delphi 实际对这些内部例程并不进行强类型检测。在编译期，内部例程的参数入口、寄存器的使用、类型检测和返回值等，编译器中都是用专门的代码来处理的。

有些时候，对于同一个函数声明，在内部需要使用多个函数来实现。这时，也会用到“compiler magic”。例如 function writeln()，在内核中，使用了超过 25 个函数来实现 writeln()，以应付不同的入口参数和自 Pascal 以来形成的古怪的语法。

此外，“compiler magic”还表明可以用内置例程来替代语言的操作符和运算符。例如在后面将提到的“PChar 类型字符串操作例程”和“集合操作例程”等例程中，就大量地使用了这种编译器支持的技术。

function Pi 也是使用“compiler magic”的一个典型例子。在 *System.pas* 中，不可能

找到 PI()的实现代码，因为这个函数是在编译期计算的。看起来，它更像是一个常量，而不是函数。下面的汇编码是将“function Pi”的结果赋给局部变量 e：

```
Unit1.pas.29: e := pi;
0044C878 C745F035C26821     mov [ebp-$10],$2168c235
0044C87F C745F4A2DA0FC9     mov [ebp-$0c],$c90fdaa2
0044C886 66C745F80040       mov word ptr [ebp-$08],$4000
```

在 Delphi 中，Pi 的十六进制值是$4000C90FDAA22168C235。上面的代码，其实就是直接修改栈上的变量 e，而不是通常理解的“将函数 Pi()的结果返回到变量 e”。

2.4.2 字符串操作例程

■ 短字符串操作例程

```
//表达式、运算和运算符
procedure _PStrCat;
procedure _PStrNCat;
procedure _PStrCpy(Dest: PShortString; Source: PShortString);
procedure _PStrNCpy(Dest: PShortString; Source: PShortString;
  MaxLen: Byte);
procedure _PStrCmp;

//操作例程
procedure _Copy( s : ShortString;
  index, count : Integer ) : ShortString);
procedure _Delete( var s : openstring;
  index, count : Integer );
procedure _Insert( source : ShortString; var s : openstring;
  index : Integer );
procedure _Pos( substr : ShortString; s : ShortString ) : Integer);
procedure _SetLength(s: PShortString; newLength: Byte);

//类型转换
procedure _SetString(s: PShortString; buffer: PChar; len: Byte);
```

基本上，这些例程都是用来替代运算符或实现运算表达式的。下例对照地进行说明：

```
var
  s1, s2, s3 : String[20];
begin
  s1 := s2;          // call procedure _PStrNCpy()
```

```
  s2 := s1 + s2;     // call procedure _PStrCpy(), and more
  s1 := s1 + s2;     // call procedure _PStrNCat()
end;
```

在上面的源码所对应的汇编码中，“s2 := s1 + s2;”是很有代表性的：

```
Unit1.pas.36: s2 := s1 + s2;  // call procedure _PStrCpy()
0044C8D5 8D55DB          lea edx,[ebp-$25]          // edx暂存s1地址
0044C8D9 8D85B4FEFFFF    lea eax,[ebp-$0000014c]    // eax暂存中间字符串(temp)地址
0044C8DF E83D61FBFF      call @PStrCpy              // temp := s1
0044C8E4 8D55C6          lea edx,[ebp-$3a]
0044C8E7 8D85B4FEFFFF    lea eax,[ebp-$0000014c]
0044C8ED B126            mov cl,$28                 // 最大长度为$28
0044C8EF E8F060FBFF      call @PStrNCat             // temp := temp+s2
0044C8F4 8D95B4FEFFFF    lea edx,[ebp-$0000014c]
0044C8FA 8D45C6          lea eax,[ebp-$3a]
0044C8FD B114            mov cl,$14                 // 最大长度为$14
0044C8FF E81C61FBFF      call @PStrNCpy             // 从temp中复制值到s2
```

这里，[ebp-$0000014c]是暂存的一个中间字符串(temp)。三次内部例程调用都进行了复制操作，而且，从 temp 到 s2 的 PStrNCpy()操作还表明：如果 s2 不够长，则在以前的代码中，“temp := temp + s2”会有一些不必要的内存复制操作。

简而言之，在 Delphi 内部实现上，短字符串是一个低效率的字符串类型——不要以为在 Delphi 中，短字符串的效率会如同它在 Pascal 中一样高。

■ 长字符串操作例程

```
//表达式、运算和运算符
procedure _LStrAsg(var dest; const source);
procedure _LStrLAsg(var dest; const source);
procedure _LStrCat(var dest: AnsiString; source: AnsiString);
procedure _LStrCat3(var dest:AnsiString; source1: AnsiString;
  source2: AnsiString);
procedure _LStrCatN(var dest:AnsiString; argCnt: Integer; ...);
procedure _LStrCmp(left: AnsiString; right: AnsiString);

//类型转换
procedure _LStrFromPCharLen(var Dest: AnsiString; Source: PAnsiChar;
  Length: Integer);
procedure _LStrFromPWCharLen(var Dest: AnsiString; Source: PWideChar;
  Length: Integer);
procedure _LStrFromChar(var Dest: AnsiString; Source: AnsiChar);
```

Delphi 源代码分析

```
procedure _LStrFromWChar(var Dest: AnsiString; Source: WideChar);
procedure _LStrFromPChar(var Dest: AnsiString; Source: PAnsiChar);
procedure _LStrFromPWChar(var Dest: AnsiString; Source: PWideChar);
procedure _LStrFromString(var Dest: AnsiString; const Source:
  ShortString);
procedure _LStrFromArray(var Dest: AnsiString; Source: PAnsiChar;
  Length: Integer);
procedure _LStrFromWArray(var Dest: AnsiString; Source: PWideChar;
  Length: Integer);
procedure _LStrFromWStr(var Dest: AnsiString; const Source: WideString);
procedure _LStrToString((var Dest: ShortString; const Source: AnsiString;
  MaxLen: Integer));
function _LStrToPChar(const s: AnsiString): PChar;

//长度计数和引用计数位的操作
function _LStrLen(const s: AnsiString): Longint;
function _LStrAddRef(var str): Pointer;
procedure _LStrClr(var S);
procedure _LStrArrayClr(var StrArray; cnt: longint);

//操作例程
procedure _LStrCopy  ( const s : AnsiString;
  index, count : Integer) : AnsiString);
procedure _LStrDelete( var s : AnsiString; index, count : Integer );
procedure _LStrInsert( const source : AnsiString; var s : AnsiString;
  index : Integer );
procedure _LStrPos( const substr : AnsiString;
  const s : AnsiString ) : Integer);
procedure _LStrSetLength( var str: AnsiString; newLength: Integer);

procedure _LStrOfChar( c: Char; count: Integer): AnsiString );
function _NewAnsiString(length: Longint): Pointer;
```

很大一部分内核字符串例程的作用，是对运算符和表达式的替换。如下例：

```
var
  GAStr1 : AnsiString;

procedure TestAnsiStr;
var
  AStr1, AStr2, AStr3 : AnsiString;
begin
  AStr1 := '';                              // call procedure _LStrClr()
  AStr1 := 'abcde';                         // call Procedure _LStrLAsg()
  GAStr1 := 'abcde';                        // call Procedure _LStrAsg()
```

```
  AStr1 := AStr1 + 'abcd';                    // call procedure _LStrCat()
  if AStr1 = GAStr1 then                      // call procedure _LStrCmp()
    AStr1 := AStr1 + 'abcd' + 'qqq';          // call procedure _LStrCatN()
end;
```

其中，向局部变量 AStr1 赋值调用的是_LStrLAsg()，而向全局变量 GAStr1 赋值时，调用的却是_LStrAsg()，这种差异是值得注意的。

此外，运算中的类型兼容和强制类型转换也会导致内部函数的调用。如下例：

```
var
  AStr : AnsiString;
  CharArr : array [0..10] of char;
  p : PChar;
begin
  //类型兼容是通过内核例程来支持的
  AStr1 := CharArr;  // call procedure _LStrFromArray()

  //(部分)类型强制转换也需要调用内核例程
  p := PChar(AStr);  // call procedure _LStrToPChar()
end;
```

部分内核函数的代码是值得仔细研读的。例如长字符串的引用计数管理实际上由两个例程来实现：

```
function _LStrAddRef(var str): Pointer;
procedure _LStrClr(var S);
```

PUREPASCAL 版本[①]的实现代码如下：

```
function _LStrAddRef(var str): Pointer;
var
  P: PStrRec;
begin
  P := Pointer(Integer(str) - sizeof(StrRec));    // 取负偏移
  if P <> nil then
    if P.refcnt >= 0 then
```

① 部分 Delphi 的内核源码同时提供两个版本，即“PUREPASCAL”和“BASM”版本。Delphi 内核是以 BASM 版本编译的——PUREPASCAL 版本的部分代码甚至在产品的正式版本中仍是有错误的。但通常而言，PUREPASCAL 版本的代码是一个非常好的对照，也是学习 BASM 最佳的 Sample。

```
    InterlockedIncrement(P.refcnt);
  Result := Pointer(str);
end;

procedure _LStrClr(var S);
var
  P: PStrRec;
begin
  if Pointer(S) <> nil then
  begin
    P := Pointer(Integer(S) - Sizeof(StrRec));    // 取负偏移
    Pointer(S) := nil;    // 字符串置空，相当于S := ''
    if P.refCnt > 0 then
      // 减一，如果减一后P.refCnt = 0，则
      // 在堆上释放P指向的内存块(即字符串的内存占用，含负偏移)
      if InterlockedDecrement(P.refCnt) = 0 then
        FreeMem(P);
  end;
end;
```

代码中涉及了 PStrRec 记录。PStrRec 记录是长字符串的内存占用的负偏移上的前 8 个字节的记录定义。定义如下：

```
type
  PStrRec = ^StrRec;
  StrRec = packed record
    refCnt: Longint;
    length: Longint;
  end;
```

因此，可以直接用 P.refcnt 来修正引用计数，也可以用 P.length 来修改字符串长度。

_LStrAddRef() 通过调用 API InterlockedIncrement() 将引用计数加 1。InterlockedIncrement()是一个原子操作函数，可以保证不会有两个操作同时作用于 P.refcnt。类似地，_LStrClr()中使用 InterlockedDecrement()来将引用计数减 1。

在 procedure _LStrLAsg()中可以见到对它们的调用：

```
procedure _LStrLAsg(var dest; const source);
var
  P: Pointer;
begin
  P := Pointer(source);
  _LStrAddRef(P);                    // 增加 source 的索引计数
```

```
  P := Pointer(dest);
  Pointer(dest) := Pointer(source); // 将 dest 的值直接指向 source
  _LStrClr(P);                      // 减少 dest 的索引计数，如果 dest 的内存空间不
                                    // 再被其他变量引用，则释放
end;
```

可以看出，长字符串赋值时，是将 dest 的值直接指向 source，并同时修正 source 和 dest 的引用计数。

■ 宽字符串操作例程

```
//表达式、运算和运算符
procedure _WStrAsg(var Dest: WideString; const Source: WideString);
procedure _WStrLAsg(var Dest: WideString; const Source: WideString);
procedure _WStrCat(var Dest: WideString; const Source: WideString);
procedure _WStrCat3(var Dest: WideString; const Source1,
  Source2: WideString);
procedure _WStrCatN(var dest:WideString; argCnt: Integer; ...);
procedure _WStrCmp(left: WideString; right: WideString);

//类型转换
procedure _WStrFromPCharLen(var Dest: WideString; Source: PAnsiChar;
  Length: Integer);
procedure _WStrFromPWCharLen(var Dest: WideString; Source: PWideChar;
  CharLength: Integer);
procedure _WStrFromChar(var Dest: WideString; Source: AnsiChar);
procedure _WStrFromWChar(var Dest: WideString; Source: WideChar);
procedure _WStrFromPChar(var Dest: WideString; Source: PAnsiChar);
procedure _WStrFromPWChar(var Dest: WideString; Source: PWideChar);
procedure _WStrFromString(var Dest: WideString;
  const Source: ShortString);
procedure _WStrFromArray(var Dest: WideString; Source: PAnsiChar;
  Length: Integer);
procedure _WStrFromWArray(var Dest: WideString; Source: PWideChar;
  Length: Integer);
procedure _WStrFromLStr(var Dest: WideString; const Source: AnsiString);
procedure _WStrToString(Dest: PShortString; const Source: WideString;
  MaxLen: Integer);
function _WStrToPWChar(const S: WideString): PWideChar;

//长度计数和引用计数位的操作
function _WStrLen(const S: WideString): Integer;
procedure _WStrClr(var S);
```

Delphi 源代码分析

```
procedure _WStrArrayClr(var StrArray; Count: Integer);
function _WStrAddRef(var str: WideString): Pointer;

//操作例程
function _WStrCopy(const S: WideString; Index, Count: Integer):
  WideString;
procedure _WStrDelete(var S: WideString; Index, Count: Integer);
procedure _WStrInsert(const Source: WideString; var Dest: WideString;
  Index: Integer);
procedure _WStrPos( const substr : WideString; const s : WideString) :
  Integer);
procedure _WStrSetLength(var S: WideString; NewLength: Integer);

function _WStrOfWChar(Ch: WideChar; Count: Integer): WideString;
function _NewWideString(CharLength: Longint): Pointer;
```

不难看出，宽字符串与长字符串的内核例程是一一对应的。

但是，前面提到过“宽字符串没有引用计数位”。那么，内部例程中_WStrAddRef()有什么意义呢？显然，必须要有新的 procedure _WStrAsg()实现方法以及虚拟的引用计数管理机制，才能适应宽字符串的内存结构。

具体的实现代码如下：

```
procedure _WStrAsg(var Dest: WideString; const Source: WideString);
asm
        { ->    EAX     Pointer to WideString   }
        {       EDX     Pointer to data         }
        TEST    EDX,EDX
        JE      _WStrClr                        // 如果Source是空串，则Dest置空串
        MOV     ECX,[EDX-4]                     // 取长度计数位
        SHR     ECX,1                           // Length(Source) div 2
        JE      _WStrClr                        // 如果结果为0，则Dest置空串
        PUSH    ECX
        PUSH    EDX
        PUSH    EAX

        // 按Source的长度重分配Dest的空间(以字为单位)，并将Source内容复制到Dest
        CALL    SysReAllocStringLen
        TEST    EAX,EAX

      // 函数调用错，内存不够，弹出系统错误：reOutOfMemory
      JE        WStrError
end;
```

```
function _WStrAddRef(var str: WideString): Pointer;
asm
        MOV       EDX,[EAX]
        TEST      EDX,EDX
        JE        @@1                       // 如果str是空串，则返回空串
        PUSH      EAX                       // pointer(str) 入栈暂存
        MOV       ECX,[EDX-4]
        SHR       ECX,1
        PUSH      ECX
        PUSH      EDX

        // 调SysAllocStringLen(str, length(str));返回新字符串地址在EAX
        CALL      SysAllocStringLen
        POP       EDX                       // 将暂存的 pointer(str) 出栈
        TEST      EAX,EAX
        JE        WStrError
        MOV       [EDX],EAX                 //使str指向SysAllocStringLen()返回的新串地址
@@1:
end;

procedure _WStrClr(var S);
asm
        { ->      EAX     Pointer to WideString  }

        MOV       EDX,[EAX]
        TEST      EDX,EDX
        JE        @@1                       // 如果是空串，则退出，否则将释放串
        MOV       DWORD PTR [EAX],0         // 置字符串的#0结束符
        PUSH      EAX
        PUSH      EDX
        CALL      SysFreeString             // 释放串
        POP       EAX
@@1:
end;
```

可以看到：

- _WStrAsg()总是将一个源字符串的拷贝赋给目标字符串，而从不执行引用计数例程_WStrAddRef()。
- _WStrAddRef()调用后，将会有一个新的字符串产生，且入口变量 Str 与返回值 Result 都指向新字符串。
- _WStrClr()总是直接释放一个非空的字符串。

因此，这些函数并没有与长字符串例程相对应的引用计数功能。Delphi 保留这些例程，从而使编译器能够以近乎同样的流程来处理两种不同的字符串[①]。

■ 未分类的字符串操作例程

```
//字符串类型转换
procedure __CToPasStr(Dest: PShortString; const Source: PChar);
procedure __CLenToPasStr(Dest: PShortString; const Source: PChar;
  MaxLen: Integer);
procedure __ArrayToPasStr(Dest: PShortString; const Source: PChar;
  Len: Integer);
procedure __PasToCStr(const Source: PShortString; const Dest: PChar);

//过程Str()和Val()
procedure _Str2Ext;
procedure _Str0Ext;
procedure _Str1Ext;
procedure _StrLong(val, width: Longint; s: PShortString);
procedure _Str0Long(val: Longint; s: PShortString);
procedure _ValExt;
function _ValLong(const s: String; var code: Integer): Longint;

//字符串唯一化操作
procedure UniqueString(var str: AnsiString); overload;
procedure UniqueString(var str: WideString); overload;
procedure _UniqueStringA(var str: AnsiString);
procedure _UniqueStringW(var str: WideString);
```

在 CStr(C 语言风格的字符串，即 Null 结尾字符串)与短字符串的相互操作中，会用到上面的内部例程。例如：

```
var
  Str : String[30];
  CStr : PChar = 'abcd';
  Arr : array [0..10] of Char;
begin
  Str := CStr;  // call procedure _CToPasStr(), and _CLenToPasStr()
  Str := Arr;   // call procedure __CLenToPasStr()
```

① 对于 Linux 系统来说，这两种字符串使用相同的处理机制。因此，Linux 版本的 WideString 例程大多数直接跳转到 AnsiString 例程中。Delphi 为 WideString 保留这些例程声明，以使得 Linux 版本的内核代码更易于维护。

```
end;
```

过程 Str()与 Val()是 Delphi 中用来在字符串与数字之间进行转换的两个例程。根据入口参数的不同，编译器会将调用的实际代码指向上面的内部例程。例如：

```
var
  s : string[200];
  e : Extended;
  I, L : Integer;
begin
  Str(12:5, s); // call procedure   _StrLong()
  Str(15, s);   // call procedure   _Str0Long()
  Str(PI, s);   // call procedure   _Str0Ext()
  Val(s, I, L); // call function    _ValLong()
  Val(s, e, i); // call procedure   _ValExt()
end;
```

字符串惟一化，是指创建一个引用计数为 1 的字符串的拷贝。四个例程的 Linux 版本都直接调用 InternalUniqueString()函数。如果是 Windows 版本，则两个宽字符串的惟一化例程是无意义的：因为 Windows 上的宽字符串没有引用计数位，这在前面已经一再讲过。

function InternalUniqueString() 的实现过程，实际上是调用 _NewAnsiString()分配新字符串并返回值。然后，将原字符串的引用减 1。如果减 1 后为 0，则释放字符串。根据 *system.pas* 中的 InternalUniqueString()改写的 PUREPASCAL 版本如下：

```
function InternalUniqueString(var str): Pointer;

  function _Ref : Integer;
  begin
    Result := PInteger(Integer(str)-8)^
  end;
  function _Len : Integer;
  begin
    Result := PInteger(Integer(str)-4)^
  end;

var
  V: String;

begin
  if PChar(str)^ = '' then
```

```
    Exit;

  if _Ref = 1 then
    Exit;

  // 模拟 _NewAnsiString() 例程
  if _Len > 0 then
  begin
    SetLength(V, _Len);
    move(Pointer(str)^, V[1], _Len);
  end;

  String(str) := V;              // 使得入口时的str引用计数减1，且变量V的引用计数加1
  Result := Pointer(Str);
end;          // 过程结束时，局部变量V的引用计数会减1，使得返回时的str的引用计数为1
```

function _UniqueStringA()是内部例程中几个不多的提供了用户调用接口的函数。用户可以调用 procedure UniqueString()，来完成相同的功能。

■ 集合操作例程

```
procedure _SetElem;
procedure _SetRange;
procedure _SetEq;
procedure _SetLe;
procedure _SetIntersect;
procedure _SetIntersect3; { BEG only }
procedure _SetUnion;
procedure _SetUnion3; { BEG only }
procedure _SetSub;
procedure _SetSub3; { BEG only }
procedure _SetExpand;
```

集合的内部例程用于实现集合运算，参见下例：

```
var
  aSet, bSet, cSet : TDemoSet;
begin
  aSet := aSet + bSet;          // call procedure _SetUnion()
  aSet := aSet * bSet;          // call procedure _SetIntersect()
  if(aSet = bSet) and           // call procedure _SetEq()
     (aSet <> cSet) and         // call procedure _SetEq()
     (cSet <= aSet) and         // call procedure _SetLe()
     (bSet >= aSet) then        // call procedure _SetLe()
```

```
  cSet := aSet - bSet;      // call procedure _SetSub()
end;
```

并非每个对应的集合操作都会调用这些内部例程。通常情况下，Delphi 试图以字节、字和双字类型的位运算来实现集合运算。并且集合运算总是优先享有使用寄存器的权利。但是，如果一个集合类型超过了 32 个元素，则编译器使用上述的内部例程来替代位运算。这意味着超过 32 个元素以后，集合运算的效率将大打折扣。

■ 数组(含动态数组)操作例程

```
// 表达式、运算和语算符
procedure _DynArrayHigh;
procedure _DynArrayClear(var a: Pointer; typeInfo: Pointer);
procedure _DynArrayLength;
procedure _DynArraySetLength;
procedure _DynArrayCopy(a: Pointer; typeInfo: Pointer;
  var Result: Pointer);
procedure _DynArrayCopyRange(a: Pointer; typeInfo: Pointer;
  index, count : Integer; var Result: Pointer);
procedure _DynArrayAsg;
procedure _DynArrayAddRef;

// 可外部调用的动态数组操作例程
procedure DynArrayClear(var a: Pointer; typeInfo: Pointer);
procedure DynArraySetLength(var a: Pointer; typeInfo: Pointer;
  dimCnt: Longint;  lengthVec: PLongint);
function DynArrayDim(typeInfo: PDynArrayTypeInfo): Integer;

// 可用于静态数组的例程
procedure _AddRefArray;
procedure _CopyArray;
```

Delphi 的动态数组采用跟字符串类同的内存结构。亦即是说，动态数组也具有引用计数位和长度计数位，也类似地采用了引用计数的机制来提高系统效率。下面的例子详细演示了内部例程的调用情况：

```
procedure DynArrayTest;
type
  TDynArr = array of char;

var
  a, a1 : TDynArr;
```

```
procedure TestDynArrayRef(const t : TDynArr);
var
  c : array of char;
begin                              // call procedure _DynArrayAddRef()
  TDynArr(c) := copy(t);           // call procedure _DynArrayCopy()
end;                               // call procedure _DynArrayClear()

begin
  if Length(a) > 0 then            // call procedure _DynArrayLength()
    a1 := a                        // call procedure _DynArrayAsg()
  else
  begin
    SetLength(a, 100);             // call procedure _DynArraySetLength()
    a1 := copy(a, 1, 3);           // call procedure _DynArrayCopyRange()
  end;

  a := nil;                        // call procedure _DynArrayClear()
  TestDynArrayRef(a1);
end;
```

动态数组的引用计数并非基于元素的，而是基于变量的，这与长字符串的引用计数不一样：在长字符串中，访问串中的一个字符，会导致引用计数发生变化；而在动态数组中，读写一个元素，并不会使引用关系发生变化——这与大多数人的设想并不一致。下例说明这一点：

```
type
  TDynArr = array of char;

procedure TestWriteCopy;
var
  a1, a2 : TDynArr;
  s1, s2 : String;
begin
  SetLength(a1, 3);
  fillchar(a1[0], 3, ORD('a'));
  s1 := 'aaa';

  a2 := a1;
  a1[1] := 'b';
  writeln(pChar(a1)); // 'aba'
  writeln(pChar(a2)); // 'aba'

  s2 := s1;
  s1[2] := 'b';
```

```
  writeln(s1);          // 'aba'
  writeln(s2);          // 'aaa'
end;
```

如果要避免修改数组元素而影响到被引用的动态数组，可以使用 function Copy()来创建一个惟一的拷贝。这与在长字符串中调用 procedure UniqueString()的效果一致。编译器会使 function Copy()直接调用内存例程_DynArrayCopy()。

```
procedure _DynArrayCopy(a: Pointer; typeInfo: Pointer; var Result:
  Pointer);
begin
  if a <> nil then
    _DynArrayCopyRange(a, typeInfo, 0, PLongint(PChar(a)-4)^, Result)
  else
    _DynArrayClear(Result, typeInfo);
end;
```

procedure _DynArrayCopy()做一个简单的检查：如果试图把 nil 赋给数组，则将结果数组 Result 清空。否则，将从动态数组负偏移上取出数组长度，并调用 _DynArrayCopyRange()。这使得下面两行代码的效果是一致的：

```
a1 := copy(a2);                    // call procedure _DynArrayCopy()
a1 := copy(a2, 0, Length(a2));     // call procedure _DynArrayCopyRange()
```

但是真正的复制代码仍然不在例程_DynArrayCopyRange()里。因为 procedure _DynArrayCopyRange()的实现代码是这样的：

```
procedure _DynArrayCopyRange(a: Pointer; typeInfo: Pointer;
  index, count : Integer; var Result: Pointer);
// ...
begin
// 1. 修正数组边界，使参数index和count有效
// 2. 从类型信息中取每个元素的长度到elSize局部变量
// 3. 在堆上分配数组空间 count*elSize + Sizeof(Longint)*2 个字节，并
//    置引用计数值为1，长度计数值为count
// ...
      if typeInf <> nil then
      begin
        FillChar(p^, count*elSize, 0);
        CopyArray(p, a, typeInf, count)
      end
      else
        Move(a^, p^, count*elSize);
```

```
// ...
end;
```

例程_DynArrayCopyRange()构造了一个新的数组。如果原数组有类型信息，则调用CopyArray()复制数组，否则，直接使用 move()操作复制数组的内容，CopyArray()调用内部例程_CopyArray()。

所有动态和静态数组，如果具有类型信息，那么它的复制操作最终都将调用到_CopyArray()。

数组的类型信息保存有一个引用列表，CopyArray()根据这个列表创建对旧数组的相应元素的引用。关于这个操作的细节，请参见“引用－计数－写复制与类型信息”小节。

Delphi 中，Copy 函数有三种声明[①]：

```
function Copy(S; Index, Count: Integer): string;
function Copy(S; Index, Count: Integer): array;
function Copy(S: array): array;
```

针对不同的数据类型，编译器会使 Copy()函数调用该类型的内部例程。如表 2-2。

表 2-2 字符串类型及调用函数

类 型	调 用 函 数
短字符串	procedure _Copy(s : ShortString; index,count : Integer): ShortString);
长字符串	procedure _LStrCopy (const s: AnsiString; index, count: Integer]: AnsiString);
宽字符串	function _WStrCopy(const S: WideString; Index, Count: Integer): WideString;
动态数组	procedure _DynArrayCopyRange(a:Pointer; typeInfo:Pointer; index,count:Integer; var Result:Pointer);

同样地，Delphi 对如下字符串例程也使用相同的技巧：

```
procedure SetLength(var S; NewLength: Integer);
procedure Delete(var S: string; Index, Count:Integer);
procedure Delete(var S: string; Index, Count:Integer);
procedure Insert(Source: string; var S: string; Index: Integer);
```

① Delphi 手册中，该函数是两种声明，第三种声明只对动态数组的唯一化有意义。

```
function Pos(Substr: string; S: string): Integer;
```

这些都可以在各自的字符串内部例程中看到对应的版本。SetLength()可以同时作用于字符串和动态数组。

■ 记录操作例程

```
procedure _AddRefRecord;
procedure _CopyRecord;
```

Delphi 的赋值运算符可以用于记录操作，编译器使该运算符指向内部例程_CopyRecord()。因此，这样的语句是可以被编译的：

```
type
  TRec = record
    I : Integer;
    S : String;
  end;
var
  R1, R2 : TRec;

begin
  R1 := R2;          // 记录赋值, call procedure _CopyRecord()
  // ...
```

同_CopyArray()一样，例程_CopyRecord()也是通过类型信息来实现的。实现的效果也类同。

有关类型信息在记录和数组中更多的应用，以及_AddRefRecord()例程的具体实现，在“引用－计数－写复制与类型信息”小节中会有更进一步的描述。

■ 其他

```
//实数类型转换
procedure _Real2Ext;
procedure _Ext2Real;

// 添加引用
procedure _AddRef;

// 浮点数除运算
procedure _FSafeDivide;
procedure _FSafeDivideR;

// Conversion utility routines for C++ convenience. Not for Delphi code.
function  CompToDouble(Value: Comp): Double; cdecl;
procedure DoubleToComp(Value: Double; var Result: Comp); cdecl;
```

```
function  CompToCurrency(Value: Comp): Currency; cdecl;
procedure CurrencyToComp(Value: Currency; var Result: Comp); cdecl;

{ Wide character support procedures and functions for C++ }
function WideCharToString(Source: PWideChar): string;
function WideCharLenToString(Source: PWideChar;
  SourceLen: Integer): string;
procedure WideCharToStrVar(Source: PWideChar; var Dest: string);
procedure WideCharLenToStrVar(Source: PWideChar; SourceLen: Integer;
  var Dest: string);
function StringToWideChar(const Source:string; Dest:PWideChar;
  DestSize:Integer): PWideChar;
```

上面例程中，_AddRef()用于对任意一个变量添加引用。看起来，它试图将变量看作一个只有一个元素的数组处理，而并不关心该元素是什么类型或者有没有类型信息。因为这个例程总是直接调用_AddRefArray()。然而事实上，这段仅仅两行代码的过程：

```
procedure        _AddRef( p: Pointer; typeInfo: Pointer);
asm
        MOV      ECX,1
        JMP      _AddRefArray
end;
```

充分利用了数组、元素和类型信息的特性，利用_AddRefArray()代码轻松地实现了对所有7种支持引用计数机制的数据类型的处理，并完成对其他数据类型的甄别。

有关浮点数运算的两个例程_FSafeDivide()和_FSafeDivideR()，可以参阅参考书目的《Delphi 技术手册》中，有关$SafeDivide 编译指令和 TestFDIV 变量的描述。

_Real2Ext()和_Ext2Real()两个例程，用于编译器在 Real 和 Extened 两种浮点数类型间相互赋值时隐含地进行类型转换。

2.5 变量的类型检测与强制转换

任何变量的类型检测都是在编译期完成的。因此，变量之间赋值，对于编译器来说，会发生如下的一些事件：

- 如果两个类型是相同的简单类型，则将源的值复制到目标；
- 如果两个类型是相同类型的记录或数组，则将简单数据类型的域复制，并添加其他域的引用计数；
- 如果两个类型是兼容的，则调用类型转换内部例程，并把转换的结果作为目标。

System.pas 中与数据类型相关的例程，主要用于支持兼容类型的转换和赋值。至于像将字符串转换为整数这样的工作，需要交给 *SysUtils.pas* 中的例程来完成。*System.pas* 中的类型转换例程是系统自动调用的，而 *SysUtils.pas* 中的转换是在代码中显式调用的。

如果使用 aType(aVar)进行强制类型转换，则只要两个数据类型大小一致，不会调用任何内部例程。数据类型的大小可通过 SizeOf()来取得。

可以使用无类型指针作为过渡来强制转换任意两个类型的变量，例如：

```
type
  // SizeOf(TRecL) = 8
  TRecL = Record
    I : Integer;
    S : String;
  end;

  // SizeOf(TArrI) = 16
  TArrI = array [1..4] of Integer;

var
  L : TRecL;
  I : TArrI;

// ...
  I := TArrI(Pointer(@L)^);
```

这种情况下，需要自行保证访问转换后的结果变量的有效性。

2.6　引用－计数－写复制与类型信息

2.6.1　引用计数与增加引用

Delphi 内部对“引用(Reference)”的处理稍显混乱。赋值运算符“=”、函数的值参数入口和一些内部例程都可能导致引用。但是相同的规则并不能应用于所有支持计数的数据类型，并且，对于一些看似类同的数据类型，规则的作用范围也不尽相同。

从内部来看，Delphi 已经尽可能地将不同类型的“增加引用”和“减少引用”例程分离开来。例如长字符串的“增加引用”调用 _LStrAddRef()，动态数组则调用 _DynArrayAddRef()。但是一些例程名字却着实让人误解，例如_AddRefArray()并非增加数组的引用，而是成组地增加一批变量的引用。

支持“引用计数”的只有少数的几个数据类型，如长字符串，动态数组，接口和变体类型。但是由于一些复杂的数据类型中的子域会声明成上述这些类型，因此，操作复杂数据类型的时候，也必须逐一考查其子域，对支持“引用计数”的数据进行计数修正。因此，虽然这些复杂数据类型也有“增加引用”的行为，但发生计数修改的是它们的子域，而非其自身。表 2-3、表 2-4 说明了这些类型的不同。

表 2-3 发生引用计数的数据类型

数据类型	例程(示例)	备注
长字符串	_LStrAddRef()	访问单一字符时，将发生写复制
宽字符串	_WStrAddRef()	限于Linux系统，Windows中，每次引用都直接复制
动态数组	_DynArrayAddRef()	引用基于变量，因此访问元素时不发生写复制
接口	_IntfAddRef()	(参考相应章节)
变体	_VarAddRef()	

表 2-4 有“增加引用”行为的数据类型

数据类型	例程(示例)	子域	备注
记录	_AddRefRecord()	域	引用计数发生在子域
静态数组	_AddRefArray()	元素	

2.6.2 “增加引用”何时发生

Delphi 中，例程中的代码可以对例程入口的值参数进行写操作。为了应付这种情况，Delphi 须在进入例程之前，为一个值参数创建一个备份。例程中对值参数的操作都实际发生在这个备份上，而不会对原来的值参数造成影响。

理论上讲，这样的备份应该是对值参数的一个完全复制。但是，Delphi 实际上只是对简单类型进行这样的处理，例如将整型值参数的值复制到堆栈，或者将集合放到寄存器。对于复杂的数据类型(例如数组、记录)，为了节约内存开销和提高系统性能，编译器对数据类型的每一个部分进行考查，并确认哪些采用“值复制”，那些采用“引用计数”，并将采用“引用计数”的域的信息附加在类型信息中的一张表里。在运行期根据这张表进行的“引用计数”操作，就是“增加引用(AddRef)”的过程。

因此，通常会看到记录和数组“增加引用”的过程发生在例程的入口处。下面的例子会在

例程的 begin 中调用_AddRefRecord()：

```
//SizeOf(TRec) = 8, 例程将在栈上开8 bytes, 用于生成一个入口值参数备份
type
  TRec = record
    I : Integer;
    S : String;
  end;

procedure TestRecordAddRef(R : TRec);
begin
  R.S := '';
end;

// ... 以下反汇编TestRecordAddRef()的begin处理

Unit1.pas.31: procedure TestRecordAddRef(R : TRec);
Unit1.pas.32: begin
0044C9BC 55              push ebp
0044C9BD 8BEC            mov ebp,esp

// 在栈上开8字节的空间, 设为_Buf, 地址为[ebp-$08]
0044C9BF 83C4F8          add esp,-$08

// 以下6条指令, 用于完成入口参数R到_Buf的复制
// 现在相当于完成了调用 move(R, _Buf, SizeOf(TRec))
0044C9C2 56              push esi
0044C9C3 57              push edi
0044C9C4 8BF0            mov esi,eax
0044C9C6 8D7DF8          lea edi,[ebp-$08]
0044C9C9 A5              movsd
0044C9CA A5              movsd

// 调用_AddRefRecord(),为记录的域增加引用
0044C9CB 8D45F8          lea eax,[ebp-$08]   // _Buf地址到eax, 开始操作备份
0044C9CE 8B15A0C94400    mov edx,[$0044c9a0]
0044C9D4 E8CB7FFBFF      call @AddRefRecord

// 以下开始在栈上填入异常帧(参见相关章节)
0044C9D9 33C0            xor eax,eax
0044C9DB 55              push ebp
0044C9DC 680ACA4400      push $0044ca0a
0044C9E1 64FF30          push dword ptr fs:[eax]
0044C9E4 648920          mov fs:[eax],esp
```

Delphi 源代码分析

```
Unit1.pas.33: // ...
```

此后，编译器会将每个对入口参数R的操作，定位到[ebp-$08]上。例如代码：

```
R.S := '';
```

会被编译为：

```
Unit1.pas.33: R.S := '';
0044C9E5 8D45FC          lea eax,[ebp-$04]   // ebp - $08 + SizeOf(P.I)
0044C9F1 E81A74F8FF      call @LStrClr
```

2.6.3 增加引用的操作是依赖类型信息来实现的

只有使用类型信息，才能在运行期分析数据类型。因此在 *System.pas* 中，数组和记录操作是为数不多的几个使用运行期类型信息(RTTI)的地方。[①]

可以通过 function TypeInfo()来取得一个指向类型信息的指针，这包括全部的基本数据类型和一些构造类型(如果它包含一些“具有引用计数特性的数据类型”的元素、字段或域，则该构造类型也具有类型信息)。和 Pi()一样，function TypeInfo()也不是一个真的函数。在编译期，Delphi 能确定每一个类型的类型信息地址，因此一旦编译器遇到 function TypeInfo()，总是直接替换成相应的类型地址。如下例：

```
Unit1.pas.29: ptr := TypeInfo(TForm);
// TProm的类型信息在地址$00441054，直接送入寄存器eax
0044C978 A154104400      mov eax,[$00441054]
// eax送入栈上的ptr
0044C97D 8945F8          mov [ebp-$08],eax
```

在_AddRefRecord()例程中就隐含了一个 typeinfo 参数，该例程实际的声明是：

```
procedure _AddRefRecord(p: Pointer; typeInfo: Pointer);
```

该例程的具体实现代码可查阅 *system.pas*，下面是根据源码改写的 PUREPASCAL 版本的 _AddRefRecord()：

```
//define in system.pas
type
```

① 绝大多数(但并不是全部)的 Delphi 基本数据类型和明确声明过的类型，都具有惟一的一个类型信息。但是像整型这样一些简单数据类型的 RTTI，通常是不需要关注的。同时需要注意的是：每一个类型信息都指向独立的内存结构。兼容的两个类型，在内存结构上也是相互独立的。

```
  TFieldInfo = packed record
    TypeInfo: PPTypeInfo;
    Offset: Cardinal;
  end;
procedure _AddRefArray(p: Pointer; typeInfo: Pointer;
  elemCount: Longint);
begin
  // code in system.pas
end;

procedure _AddRefRecord(const R; typeInfo: Pointer);
var
  i, maxI : Integer;
  p : Pointer;
begin
  inc(PByte(typeInfo));                          // skip kind
  inc(PByte(typeInfo), Byte(typeInfo^)+1);       // skip name(ShortString)

  inc(PByte(typeInfo), 4);
  maxI := PInteger(typeInfo)^;                   // 取引用表(数组)的长度

  inc(PByte(typeInfo), 4);                       // 定位到引用表的头部
  for i := 0 to maxI-1 do
  begin
    p := Pointer(Integer(@R) + TFieldInfo(typeInfo^).Offset);
    _AddRefArray(p, TFieldInfo(typeInfo^).TypeInfo, 1);
  end;
end;
```

_AddRefRecord()总是在记录的 typeInfo 中找出引用列表(数组)的长度和表首地址。引用列表(数组)只保存需要定位的类型变量的信息，它的元素类型为 TFieldInfo。TFieldInfo.Offset 表示需要定位的元素自变量首地址开始的偏移字节数。因此，对于如下的记录：

```
type
  TRec = record
    I : Integer;
    S : String;
  end;
```

来说，引用列表(数组)只有一个元素，其 FieldInfo.TypeInfo 指向 TRec.S 的类型信息(即 String 的类型信息)，而 FieldInfo.Offset 值为 4，表明 TRec.S 的地址相对于记录首部的偏移为 4 Byte。

整型是简单数据类型，所以 TRec.I 直接复制值；而 String 类型使用引用计数机制，所以 TRec.S 被加入引用列表(数组)——这是它们的不同。

2.6.4 写复制与值参数的备份

写复制技术被用在很多的地方，例如 DLL 的映像，但这是在运行期由操作系统决定的行为。在 Delphi 中，写复制是在编译期就被决定的，它通过内核例程和编译器的语义分析两个方面来实现。

只有使用引用计数的数据类型会发生写复制。具体的说，是指长字符串、宽字符串、动态数组、接口和变体这五种数据类型。写复制机制内嵌于类型的基本操作例程中，且只有在数据发生修改时，写复制才会发生。

以长字符串为例。编译器在语义分析时，会检查每一段使用下标访问字符串的语句，并在最前面插入“call @UniqueStringA”。这个例程用来检查字符串的引用计数，如果发现字符串被引用(引用计数大于 1)，就复制一个新的字符串并返回给原变量。这样，操作这个字符串就不会再影响到引用它的其他字符串。

但是，字符串的内部例程并不调用_UniqueStringA()。例如_LStrCat()，该例程会首先调用_LStrSetLength()来为结果字符串分配内存，而_LStrSetLength()则先检测引用计数位，如果大于 1，则调用_NewAnsiString()来创建新的字符串。源代码如下：

```
procedure       _LStrSetLength( var str: AnsiString; newLength: Integer);
asm
        // ...
        CMP       [EAX-skew].StrRec.refCnt,1
        JNE       @@copyString
        // ...
@@copyString:
        MOV       EAX,EDX
        CALL      _NewAnsiString
        // ...
end;
```

值参数的备份源自于例程的调用约定：值参数可以被改写，但是不会影响到传值的原始变量。因此，编译器需要为值参数制作一个备份，而无论代码中是否要写该值参数。这个复制操作在编译时就被决定了，例程被调用一次，就发生一次复制操作。这与写复制是不同的。

前面已经讲述过，在值参数备份中，针对不同的数据类型，生成备份的方法不同，具体如下：

- 如果是简单数据类型，则直接复制参数的值。

- 如果是具有引用计数位的数据类型，则调用相应的函数产生一个引用。
- 如果是数组或记录，则将元素或域按照上述规则分别处理。

■ 其他

数组和记录的引用列表(数组)其实也是它们的初始化表。在第 4 章的“初始化与结束化过程”一节中，会再一次讲到它们。

如果确知在代码中不需要改写入口的值参数，那么可以用 const 声明它。这样做的结果是编译器将向例程直接传入参数的地址，而不再是使用参数值的引用。正是因此，如果使用指针访问该地址，就可以对原值进行直接修改。下面的例子揭示如何修改用 const 声明的值参数：

```
{$O-}
{$APPTYPE CONSOLE}
type
  PRec = ^TRec;
  TRec = record
    I : Integer;
    S : String;
  end;

procedure ChangeConstParam(const R:TRec);
begin
  PRec(@R)^.I := 6;
end;

var
  Rec : TRec;
begin
  Rec.I := 5;
  writeln(Rec.I);

  ChangeConstParam(Rec);
  writeln(Rec.I);
end.
```

第 3 章

BASM(Borland 汇编语言)精要

简单地了解一些 BASM[①]的知识，对于阅读 Delphi 内核源码是很有必要的。

与 Delphi1-5 版本不同，高版本 Delphi 正在努力减少内核源码中汇编所占的比例。似乎 Borland 越来越想证明：即使不用汇编，也可以完成编译语言的全部内核。

BASM 支持指令包括：

- Pentium 系列
- Pentium Pro
- MMX
- SIMD
- SSE
- AMD 3DNOW!

这些内容请自行查阅相关的资料，本节内容不讲述基本语法与指令系统，而是着重介绍 Delphi 代码与 BASM 代码的交互使用，以及调用约定。

此外，BASM 对标准 ASM 的补充和扩展也是本节讲述的一个重点。

3.1 BASM 概念简要

汇编语句由指令和零至三个表达式构成。表达式由常数(立即数)、寄存器和标识符构成。例如：

① BASM 的官方解释为 Built-in assembly，在 Delphi 中的 BASM 也被为 Delphi inline assembly。BASM 被解释作“Borland 汇编语言(Borland assembly)”只是一些程序员的习惯。

```
movsb          // 单指令语句
jmp @Here      // 一个表达式：标识符
add eax,1      // 两个表达式：寄存器和立即数

// 三个表达式：寄存器，标识符(内存地址)，立即数
imul edx, [ebx].RandSeed, 08088405H
```

一段 BASM 代码以 ASM 关键字开始，END 关键字结束。中间有任意多个汇编语句。

BASM 代码通常写在例程中。Delphi 的 BASM 是内嵌于语言的，无法独立编译出可执行程序或中间代码(*.Obj*)。但是，可以使用 BASM 来完成一个完全汇编的程序，并使用 Delphi 编译器编译。如下例：

```
program TestBASM;
asm
  mov eax, 100
end;
```

3.2　表达式的类别与类型

在 BASM 的语句中，每一个表达式都必须能够在编译器中计算出准确的值或者寻址地址。如果不能满足这个条件，语句不会被编译通过。事实上，对于指令系统来说，每一个表达式都最终对应于一个确定的操作数。

因此，表达式的类别(Expression classes)，按表达式的计算结果可分成三类：寄存器、立即数和内存引用(存储器)。

与内存引用相关的表达式，会涉及到存储器寻址模式的问题，请查阅相关资料。下一小节会简要讲述在 BASM 中访问 Delphi 所定义的变量与常量，但不涉及寻址模式。

在 BASM 中，表达式的类型(Expression types)是一个长度值，它是指表达式值占用空间的字节数，即值的大小。这与 Delphi 中 `SizeOf()`函数含义是一样的。但 BASM 中用关键字 TYPE 来返回表达式的类型(大小)。

如下例：

```
type
  TArr = array [0..10] of char;  // SizeOf(TArr) = 11

var
  Arr : TArr
```

```
asm
  mov eax, TYPE Arr
  mov eax, TYPE TArr
  mov eax, TYPE Arr[2]
end;
```

上面的三行汇编语句都会向 eax 送入值 11。第三行看起来是要取 Arr 数组元素的长度，但实际上只能取到数组的长度。

较为复杂的表达式，其类型由第一个操作数的类型来决定。因此下面这个语句送入 eax 的值仍然为 Arr 的类型值 11：

```
mov eax, TYPE (Arr + 2)
```

这里的括号不能理解成函数，而是用来改变运算优先级的。

同样的道理，在 BASM 中，以下两条语句面对的命运是不同的：

```
mov eax, 2 + Arr
mov eax, Arr + 2
```

第一代码行会被 BASM 理解成 Arr 的地址值+2。而第二行代码右边表达式的长度为 11，不能送入寄存器 eax，因而根本不会被编译通过。

3.3 数据定义和数据类型强制转换

BASM 可以使用所有通过 Delphi 语法定义的变量、常量。BASM 扩展了 ASM 的语法，用于访问记录、数组、对象等复杂的数据结构。

下例简单解释了如何进行数据定义和访问：

```
type
  TRec = record
    rI : Integer;
    rS : String;
  end;

var
  I : Integer;
  R : TRec;
  S : String = '1234567';
```

```
  A : Array [0..10] of char
    = 'abcdefghij'#0;
const
  C = 3124;
  Str = 'abcde';

asm
  mov eax, I                          // I 的值送入 eax
  mov eax, [I]                        // 同上
  mov eax, OFFSET I                   // I 的地址送入eax, 相当于 eax = @I

  mov eax, R.rI                       // 域rI的值送入eax
  mov eax, [TRec.rI + R]              // 同上
  mov eax, [Offset R + TRec.rI]       // 同上

  mov ebx, S
  dec ebx                             // 忽略s[0]
  mov esi, 4
  mov al, BYTE [ebx + esi]            // 将s[4]的字符值送入al
  mov al, BYTE [ebx + 4]              // 同上
  mov eax, [ebx+4]        // 将s[4]..s[7]四字节以DWORD值送入eax, eax=$37363534

  mov ebx, OFFSET A
  mov eax, [ebx+4]        // 将 A[4]..S[7]四字节以DWORD值送入eax, eax=$68676665

  mov eax, C              // eax = 3124
  mov eax, [C]            // eax = PInteger(3124)^, 非法的内存地址访问
end;
```

在上例中，常量 C 总是作为数值直接被编码。因此，“mov eax, C”中，它作为立即数 3124 被送入 EAX。而在“mov eax, [C]”却表明要访问内存地址“3124”，因为“[C]”表明是内存引用。

由于常量总是被直接编码，上例中，无法访问常量 Str——Str 的长度大于 4，所以无法送入 EAX。同样的原因，在 BASM 中，对常量使用 OFFSET 是没有意义的——尽管在 Delphi 中，字符串常量可以具有内存地址。下例中，EAX 总是被送入 Str 的值，而非地址。

```
const
  Str = 'abcd';
  Str2 = 'ab';

asm
  // eax = $61626364, OFFSET是无意义的
  mov eax, OFFSET Str
```

Delphi 源代码分析

```
  // eax = $00006162, 如果字符串长不大于4，可以送入eax.长度不够时，在左侧补0
  mov eax, Str[2]
end;
```

BASM 不支持访问数组下标(可以用地址运算来替代这样的语法)。尽管类似“mov eax, TYPE Arr[2]”这样的语句可以编译通过，但它总是返回数组的整个长度(如上一节例子中的值 11)。这也正好解释了“mov al, Arr[2]”这样的语句为什么不能被编译——因为要将一个类型长度为 11 的数据放入 al 寄存器，是无法做到的。

BASM 中支持两种类型强制转换的语法，效果是完全一致的。

```
type
  TCode = Record
  I : Integer;
  S : String;
end;

var
  aRec : TCode;
  aInt : Integer;

asm
  mov eax, aInt:TCode.I          // 使用“表达式.类型”的强制转换格式
  mov eax, integer(aRec)         // 使用“类型(表达式)”的强制转换格式
end;
```

这里的强制转换的语义与 Delphi 是一样的。但是，BASM 的强制转换，只是把地址上的变量强制识别成目标类型，而不进行长度校验。因此可以看到，TCode 的长度为 8，而整型长度为 4，它们之间仍然可以转换，这样的转换在 Delphi 中是行不通的。

BASM 代码块中，也可以定义数据。但是，用 BASM 语句定义的数据总是在代码段里，这也是对 Delphi 无法在代码段里定义数据的一个弥补。

BASM 支持四个用于定义数据的汇编指令 DB/DW/DD/DQ。与 ASM 不同，不能为这些数据命名。例如：

```
asm
      DB 0FEH          // 定义一个字节
  aVar DB 0EFH         // 在ASM中可用，但在BASM中不支持
end;
```

可以通过一些技巧来解决命名问题。但是，必须同时用操作系统的 API 来打开代码访问权限，才能真正的写这些数据。下面的例子展示数据定义、命名和读取的方法：

```
type
  TCode = packed Record
    CODE : WORD; // jmp @, 2 Bytes
    I    : Integer;
    S1   : array [1..26] of char;
    S2   : array [1..11] of byte;
  end;

var
  I : Integer;
  S : String;
  Code : ^TCode;

function ReadCode : Integer;
asm
  jmp @
  DD 12344213
  DB 'ABCDEFGHIJKLMJNOQRSTUVWXYZ'
  DB 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32
@:
  mov Code, offset ReadCode
  mov EAX, ReadCode.TCode.I
end;

// ...
  I := ReadCode;      // I = 12344213
  S := Code^.S1;      // S = 'ABCDEFGHIJKLMJNOQRSTUVWXYZ'
```

这个例子以例程名作为变量的地址，但并不是一个好的例子(尽管很多代码这样做)。更方便的方法是使用标号作为变量名，与上例类同的例子是这样：

```
type
  TCode = packed Record
    I    : Integer;
    // ...
  end;

var
  I : Integer;

function ReadCode : Integer;
asm
  jmp @
@CodeRec :
  DD 12344213
```

```
  // ...
@:
  mov EAX, @CodeRec.TCode.I  // 使用标号作为变量
end;

// ...
  I := ReadCode;   // I = 12344213
```

3.4 例程入口参数及调用约定

任何情况下，在寄存器的使用上，BASM 遵循如下的规则：

- ASM 语句执行过程中，必须保存 EDI、ESI、ESP、EBP、EBX 的值。
- ASM 语句可以任意使用 EAX、ECX、EDX。
- 一个 ASM 代码块开始时，EBP 指向当前堆栈，ESP 指向栈顶。
- SS 存放堆栈段的段地址；DS 存放数据段的段地址；CS 存放代码段的段地址。
- 通常情况下，段地址寄存器满足如下条件：SS=ES=DS。
- 如果需要，函数总是以 EAX(32 位)、AX(16 位)或 AL(8 位)作为返回值的寄存器。

Delphi 的例程入口参数有以下几种：

```
procedure TestProc(I : Integer);          // 值参数
procedure TestProc(var I : Integer);      // 变量参数
procedure TestProc(const I : Integer);    // 常数参数
procedure TestProc(out I : Integer);      // 输出参数
```

按照 Delphi 的语法规定，值参数和常数参数使用相同的传值规则，但值参数只是传入值的备份；变量参数、输出参数总是传入值的地址。至于像“无类型参数”、“开放数组参数”等，都是在上面的基础上声明的，因此也符合其基本规则。

可以直接修改变量参数和输出参数传入的内存地址上的值，这种修改能被调用者识别和接收。

对于值参数，必要的情况下，编译器会生成一段代码，用于创建值参数的一个备份并用它的地址替换入口参数的地址。除此之外，值参数与常数参数使用相同规则：如果传入的数据长度小于或等于 4 Bytes(这存在一些例外，如 Int64)，则直接传值，否则传值的(对于值参数来说，是值的备份的)内存地址。

在不违背上述寄存器使用规则和例程参数传递规则的前提下，Delphi 支持 5 种调用约定（如表 3-1 所列）。

表 3-1　例程调用约定

调用约定	传参顺序	清除参数责任	寄存器传参	实现目的	其　他
register	由左至右	例程自身	是[1]	提高效率	Delphi默认规则。类设计中，公开的声明强制使用该约定
pascal	由左至右	例程自身	否	与旧有过程兼容	较少使用
cdecl	由右至左	调用者	否	与C/C++模块交互	Powerbuilder等其他语言也使用该约定
stdcall	由右至左	例程自身	否	Windows API	Windows API通常使用该约定
safecall	由右至左	例程自身	否	Windows API，COM	用于实现COM的双重接口，错误与异常处理

3.5　例程和 API 的调用与流程控制

根据调用约定，通常以 register 约定来调用 Delphi 的函数和过程，以 cdecl 约定来与其他语言混合编程，以 stdcall 约定来调用 Windows 的 API。

下面的例子演示如何调用 Delphi 的函数：

```
function DelphiFunc(I: Integer; var S1, S2:String) : Integer;
begin
  if I < Length(S1) then
    SetLength(S1, I);
  S2 := S1 + S2;
  Result := Length(S1);
end;

var
  GS : String = '12345678';
procedure RegisterCall;
var
  LS : String;
  Len : Integer;
```

① 使用寄存器传递参数时，EAX、EDX、ECX 分别存放第一、二、三个参数。如果有更多的参数，将使用堆栈传递参数。参数是否以寄存器传递，取决于例程中声明参数的类型、顺序和个数。

```
begin
  LS := 'This is a test!';

  //以下汇编代码相当于Delphi语句
  //  Len := DelphiFunc(8, LS, GS);
  asm
    mov eax, 8
    lea edx, LS              // 传入局部变量 LS. 局部变量必须使用lea指令载入地址
    mov ecx, OFFSET &GS      // 传入全局变量 GS. 变量名与BASM保留字中的GS(段地址寄存器)
                             // 冲突, 因此加复写标识符"&". 也可以使用语句lea ecx, &GS
    call DelphiFunc
    mov Len, eax
  end;

  writeln(LS);               // 'This is 12345678'
  writeln(Len);              // 16
end;

  // ...
  RegisterCall;              // 调用该例程, 显示局部变量LS和Len的值
```

下面的例子演示如何调用 Windows API:

```
function GetFileSize(Handle: Integer; x: Integer): Integer; stdcall;
  external 'kernel32.dll' name 'GetFileSize';

function stdcallDemo : Integer;
var
  FH : THandle;
begin
  FH := FileOpen('C:\boot.ini', fmOpenRead);

  //以下汇编代码相当于Delphi语句
  //  Result := GetFileSize(FH, nil);
  asm
    push 0                   // 第二个参数 nil 入栈
    push FH                  // 第一个参数 FH 入栈
    call GetFileSize         // 依据stdcall约定, 例程GetFileSize()将清理栈, 所以BASM
                             // 中不考虑nil和FH参数的出栈
    mov @Result, eax         // 按约定, 返回值在eax中, 将eax值送入stdcallDemo()的返回值.
                             // @Result由BASM定义
  end;

  FileClose(FH);
```

```
end;

  // ...
  writeln(stdcallDemo);  // 输出文件'c:\boot.ini'的长度
```

可能的情况下，BASM 总是试图调整跳转指令，尽可能地使用短程跳转(2 Bytes)，否则使用近程跳转(3 Bytes)。只有在两者都不可能的情况下，才会使用远程跳转(5～6 Bytes)。此外，如果是远程条件跳转指令，例如：

```
JC FarJump
```

BASM 会将指令转换成这样的形式：

```
  JNC ShortJump
  JMP FarJump
ShortJump:
  // next line ...
```

BASM 中，可以用跳转指令将流程指向当前单元中的任何例程。这使得一些错误控制更加简单而且高效。例如 *System.pas* 中，试图调用纯虚方法时会进入例程_AbstractErro()，这时，_AbstractError()会使用一个 JMP 跳转到系统的错误处理例程_RunError()：

```
@@NoAbstErrProc:
        MOV       EAX, 210
        JMP       _RunError
```

使用 JMP，而不是 CALL 的区别在于：JMP 跳转使得目标例程替代了当前例程的 RET 指令，这样，在错误处理后，出错点的后续指令将不会再被执行。如图 3.1 所示。

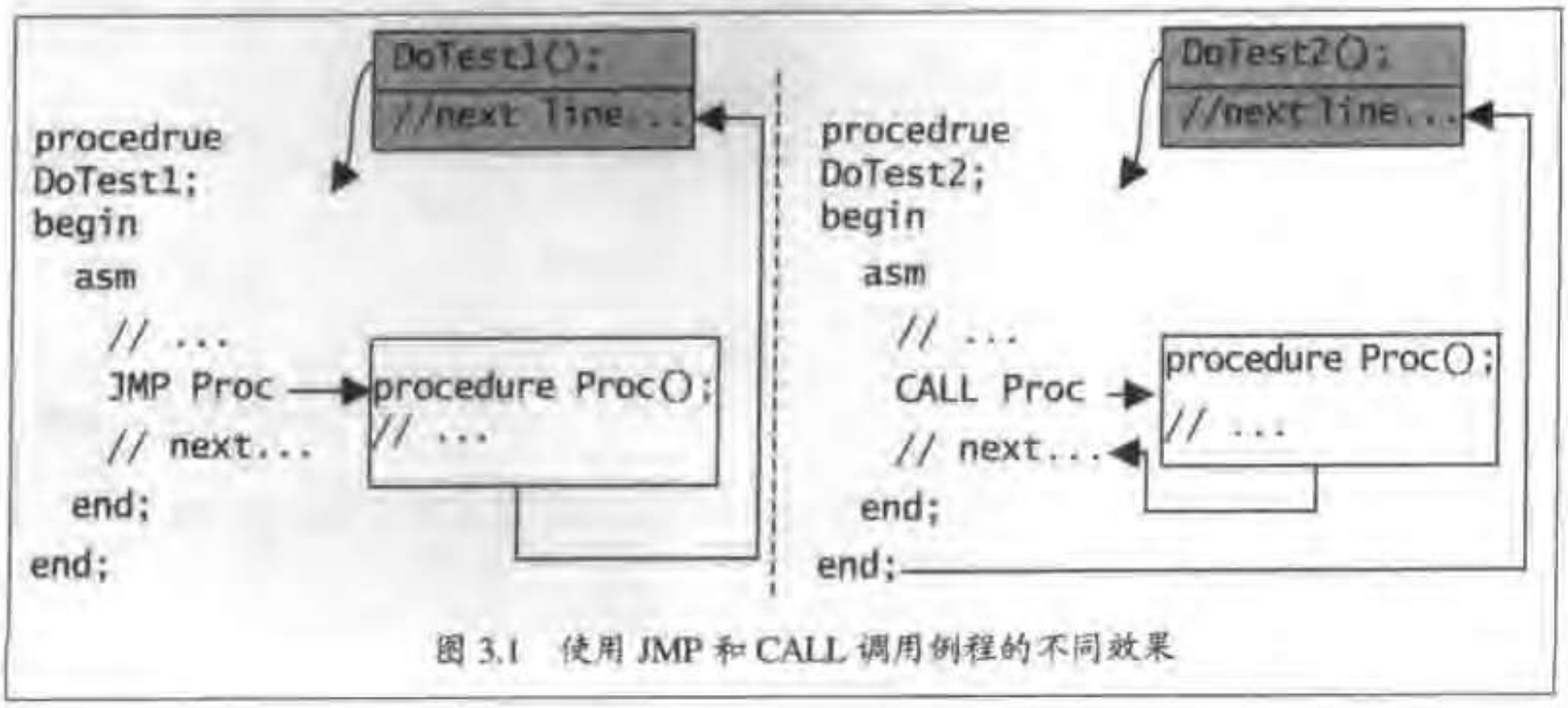


图 3.1　使用 JMP 和 CALL 调用例程的不同效果

如果要使 JMP 指令跳转返回到下一行，那么，可以用类似下面的技巧修改 EIP 指针来实现：

```
DB $E8, $0, $0, $0, $0, $8F, $04, $24, $83, $04, $24, $0C
jmp proc
```

在 BASM 中的任意位置加入上述代码，即可使得“jmp proc”执行后返回到下一行。上面用 DB 定义的内嵌汇编代码的实际代码如下：

```
// ...
  call @@GetEIP // $E800000000, 将标号@@GetEIP位置作为过程入口调用@@GetEIP;
  pop [esp]     // $8F0424, 3字节，从栈顶弹出EIP值到[esp]，该值为@@GetEIP标
                // 号的地址
  add [esp], 12 // $83042400C, 4字节，在@@GetEIP地址上加12个字节，作为真实的返
                // 回地址在@@GetEIP和@@ReturnHere之间的三条指令长度总是为3+4+5
                // =12 Bytes
  jmp proc      // 无条件远程跳转，长度为5字节
@@ReturnHere:
```

也就是说，“jmp proc”跳转到的目标例程返回(RET)时，使用的将是“Call @@GetEIP”时入栈的 EIP 值，而这个 EIP 值又通过“+12”被修改成@@ReturnHere 的地址。因此，“jmp proc”总是返回到@@ReturnHere 位置，从而得到了与“call proc”类同的效果[①]。

3.6 完全汇编例程与内嵌汇编例程

BASM 在例程中使用时，可以分成完全汇编例程和内嵌汇编例程两种。完全汇编是指用 asm 关键字替换了例程的 begin，从而使例程完全由汇编代码实现。在 Begin..End 中间任意位置加入 asm..end 的 Delphi 例程都称为内嵌汇编例程。

完全汇编例程中没有例程入口时的 begin，因此，Delphi 不会形成值参数的复制。这意味着在完全汇编例程中，值参数与常数参数的处理是一致的。

通常情况下，编译器会自动处理例程的堆栈结构。但是，如果完全汇编例程不是一个子例程(例程嵌套)，也没有入口参数(或它们只占用寄存器)和局部变量，则编译器不会为该例程产生堆栈结构。亦即是说，这样的例程不会在堆栈上分配空间。

完全汇编例程的 asm 关键字会被编译器解释成例程入口代码。例如：

① 这样的技巧可以用于加密子过程，也可以实现在操作系统或应用程序模块中“无缝地”钩挂(HOOK)代码。

```
Unit1.pas.34: asm
0044C86C 55              push ebp
0044C86D 8BEC            mov ebp,esp
```

只要定义了局部变量，或入口参数使用到了栈，则会生成上面的代码。但是，局部变量定义还会导致类似这样的一行代码产生：

```
0044C86F 83C4D8          add esp,-$28
```

这行代码用于在栈上为局部变量分配空间(本例中是$28 Bytes)。但是，如果所有变量在栈上分配的总空间不大于4字节，那么编译器会处理成：

```
0044C86F 51              push ecx
```

这样实际上也使esp调整了4字节。但效率会比“add esp，-$4”要好得多。

如果局部变量是字符串、变体或接口类型，那么这些变量会被初始化为0。因此，这样的情况下，编译器通常采用“push $00”的方式来实现空间分配。而在一些复杂的情况下，编译器会直接写栈来初始化这些变量，例如：

```
0044C872 33C0            xor eax,eax
0044C874 8945FC          mov [ebp-$04],eax
```

对应于在入口代码中加入的“push ebp”，代码出口处，编译器会生成“pop ebp”。

除了上述的这些情况之外，编译器不会为完全汇编例程加入其他多余的代码。

如果需要在例程中加入局部变量，但又不影响堆栈，可以使用在例程中定义类型化常量的方法，来代替变量声明。

3.7　汇编例程中的返回值约定

在完全汇编例程中，函数必须按如下的规则来返回值[①]：

- 按照数据类型的长度，序数类型和一些简单类型(例如集合)使用AL、AX或EAX返回。
- 实数类型通过浮点运算器的寄存器堆栈的ST(0)返回。Currency类型须先放大10000倍。
- 指针类型、类类型以及类引用类型使用EAX返回。
- 对于字符串、动态数组、方法指针、变体以及其他一些大小超过4字节的数据类型(例

① 这事实上也是Delph编译器处理函数返回时的一般规则。

如短字符串、变体等)的返回值来说，返回值是通过在函数声明的参数之后另外传入的变量参数返回的。

对于最后一条规则，开发人员通常并不需要计算 Delphi 将如何“另外传入一个变量参数”，而只需要在汇编代码中通过@Result 返回值即可——Delphi 会按照上述的规则完成编译。

对于内嵌汇编例程来说，上面的规则完全不适用——编译器将按 Delphi 的规则为例程的关键字“Begin .. End”生成入口与出口的处理代码，返回值也由例程(而非内嵌汇编代码)处理。而且，在使用 Registry 调用约定的例程的内嵌汇编代码中，EAX、EDX 和 ECX 未必总是例程入口参数的前三个——因为例程的其他代码可能已经重写了这些寄存器。

3.8 其他

下面整理了一些在 BASM 编程过程中常用资料的速查表格。

■ BASM 的保留字

表 3-2 列出了 BASM 支持的保留字和运算符以及其他一些符号。其中“@”是用做标号，“,”是分隔符

这些保留字的详细描述参见后续表格。

表 3-2 BASM 支持的保留字和运算符以及其他符号

寄存器	EAX, AX, AH, AL EBX, BX, BH, BL ECX, CX, CH, CL EDX, DX, DH, DL DS, CS, SS, ES, GS, FS ESI, EDI, SI, DI, EBP, ESP, BP, SP mm0..mm9, xmm0..xmm7 ST, EIP(保留但无法直接使用)
类型标识符	BYTE, WORD, DWORD, QWORD, TBYTE
操作符	AND, OR, XOR, NOT, MOD, LOW, HIGH, OFFSET, PTR, TYPE, SHL, SHR
	LARGE, SMALL, DMTINDEX, VMTOFFSET
运算符	“+”，“-”，“*”，“/”，“.”，“:”，“&”，(...)和[...]
其它符号	“@”、“,”

■ BASM 支持的运算(运算符和操作符)

表 3-3 列出了 BASM 支持的运算符和操作符，以及他们的优先级。在注解中还包括了使用这些运算时的简要示例。

表 3-3　BASM 支持的运算符和操作符

运算符	注　解	优先级
&	覆盖内存标识符。下例表明HIGH是用户自定的一个标识符，而非保留字： MOV　EDX,[&HIGH]	高
(...)	指定优先运算的子表达式，与Delphi中的(...)用法相同。	-1
[...]	将方括号内的表达式运算结果视为内存地址。包括： 1、由EBX,EBP,ESI,EDI等寄存器、"+"和立即数构成的寄存器索引寻址 2、表达式(或其运算结果)的地址 例如，把地址[EBX-1]上的一个整型值送入EDX： MOV　EDX, [EBX-1]	
.	访问记录的域或对象的字段、属性与方法①。例如操作记录： MOV　EAX, [EAX].vmtParent 或者作为类型强制转换。例如将[EAX]转换成TTextRec： CALL　[EAX].TTextRec.InOutFunc 或指定引用单元。例如调用SysInit.pas中的例程： CALL　SysInit.@GetTLS	
HIGH, LOW	返回表达式结果的高(HIGH)或低(LOW)八位。例如： MOV　AX, LOW aObjInstance	
+,-	一元加法或者减法运算。例如在栈上分配临时变量： ADD　ESP, -528	-2
:	段引用，用来为其后的表达式(作为地址值)指定段。例如在异常处理中： MOV　FS:[EAX], ESP	-3
OFFSET	返回一个立即数，表明运算符之后的表达式的位移部分。例如异常处理中： PUSH　offset @@returnAddress	-4
VMTOFFSET	取(对象或类)虚方法的偏移值。例如调用[ECX]中的对象虚方法Destroy： CALL　DWORD PTR [ECX] + VMTOFFSET TObject.Destroy 或： CALL　DWORD PTR [ECX + VMTOFFSET TObject.Destroy]	

① 这样的说法并不确切，只是更加容易理解而已。官方资料中的说明是这样的："句点符是'结构成员选择运算符'，整个结果是句点之前与之后两个表达式的和，并以句点之后的表达式类型为准。"因此按照官方的说法，则在这个例子中，TTextRec.InOutFunc 应该理解为一个记录结构的偏移值，而[EAX].TTextRec.InOutFunc 则应当理解为"计算记录头部地址([EAX])与 InOutFunc 域的偏移之和，并返回该位置上的值，类型以 InOutFunc 域的值为准"。注意，对象是按照"结构(亦即是记录)"的规则来处理的。

续表

运算符	注 解	优先级
DMTINDEX	取(对象或类)的动态方法的索引值。例如调用[ECX]中的TComponent.ExecuteAction()[①]: MOV EAX, ECX MOV ESI, DMTINDEX TComponent.ExecuteAction CALL System.@CallDynaInst	-4
TYPE	取表达式的类型值(大小)。	
PTR	指明表达式的结果地址指向某种类型。例如以PByte类型操作地址[EDX+ECX]: MOV byte ptr [EDX+ECX],0	
*, /, MOD	二元乘、除法运算,取模运算。	
SHL, SHR	逻辑左移、右移运算。	
+, -	二元加、减法运算。	
NOT, AND, OR, XOR	位运算。	最低

■ BASM 支持的类型声明指令和类型标识符

在表 3-4 中只示范了 TBYTE 这个标识符的使用,可以作为其它标识符使用时的参考(BASM 中 TBYTE 类型没有相应的声明指令)。此外,立即数是不需要用标识符声明的,其类型值为 0。

表 3-4 标识符 TBYTE 的使用

标识符	类型	声明指令(示例)	标识符使用(示例)
BYTE	1	asm DB $20,$0 DW $0020,$20 DD $FFFFFFFF,$0 DQ -1,$2 end;	var X : Extended; // TBYTE function CalcSQRT : Extended; asm FLD TBYTE PTR [X] FSQRT end;
WORD	2		
DWORD	4		
QWORD	8		
TBYTE	10		

■ BASM 支持的由 Delphi 声明的标识符

在 Delphi 的 Object Pascal 代码中声明的一些标识符是可以被 BASM 识别的,它们以特定的

① 在 BASM 中调用动态方法时,对象实例地址应送入 EAX,动态方法在 DMT 表中的索引应送入 DSI 或 SI,然后调用内部例程_CallDynaInst()。此外,DMTINDEX 和 VMTOFFSET 运算符的优先级值来自于非官方资料。

值(以及类型)参与运算。表 3-5 给出了这些标识符。

表 3-5　Delphi 的 Object Pascal 代码中声明的可以被 BASM 识别的一些标识符

标识符	内容	类别	类型	说明
字段和域(Field)	偏移值	内存	数据类型的大小	这里返回的不是内存地址，而是一个用于运算的偏移值。所以"类别"不是内存引用，而是内存(更确切地说，是立即数)。
变量(Var)	变量的Address	内存引用	数据类型的大小	
标号(Label)	标号的Address		短程地址(SHORT)	由于是内存引用地址，因此可以进行地址运算或函数调用(JMP/CALL)。"类型"中SHORT和NEAR的大小与系统的CPU有关(例如64 bits)。
过程(Procedure)	过程的Address		近程地址(NEAR)	
函数(Function)	函数的Address			
常数(Const)	常数值	立即数	0	类型化常量的"类型"值是例外。
数据类型(Type)	0		数据类型的大小	
程序单元(Unit)	0		0	单元名可作为标识符直接参与运算。
@Result[1]	Result变量的偏移值	内存引用	数据类型的大小	参见本章第7节"汇编例程中的返回值约定"。

在 BASM 中也可以访问 Delphi 的一些内部例程的标识符，但得到的并不是这些例程的地址，而(可能)是它们的 compiler magic identity[2]。如下例：

```
asm
   mov edx, SizeOf
   mov edx, TypeOf
   mov edx, TypeInfo
end;
```

在 CPU 窗口中看到的结果是：

```
Project2.dpr.8: mov edx, SizeOf
004033A4 BA32000000      mov edx,$00000032
Project2.dpr.9: mov edx, TypeOf
004033A9 BA39000000      mov edx,$00000039
Project2.dpr.10: mov edx, TypeInfo
004033AE BA38000000      mov edx,$00000038
```

① 在 Delphi 6 官方的《Object Pascal 语言手册》中，还介绍了@Code 和@Data 两个标识符。但这是不正确的，这两个标识符和另一个段运算符(SEG)都是 Delphi 1 中的过时内容，从 Delphi 2 开始，它们就已经完全消失了。

② 这些值可能是编译器使用的一个内部编号，虽然目前无法确定它的具体特性，但它的确在 Delphi 中存在。

第 4 章

初始化与结束化过程

每一个代码块都具有一个 BEGIN 和一个 END。尽管在 BASM 中用 ASM 关键字代替了 BEGIN，但它仍然起着相同的作用。

有些情况下，BEGIN 和 END 只决定一个语法结构的开始和结束，例如记录和对象类型定义中使用的 BEGIN 和 END。但更多的时候(例如例程定义中)，BEGIN 与 END 代表初始化与结束化过程。这一切都是编译器完成的，大多数情况下都是不可见的。

Delphi 中加入了 initialization 和 finalization 关键字来定义单元的初始化和结束化节。可以用 begin 或 asm 关键字替代 initialization——一惟一的影响是：不能再定义 finalization 节。

Delphi 会为每个单元加入一个 initialization 节，即使没有显式定义。

项目文件(*.DPR*)没有 initialization 与 finalization 节，它的 begin..end 代表了模块的载入与卸载过程。包的项目文件(*.DPK*)不能定义 begin..end 块，它的 end 关键字指向的过程同时担任了载入和卸载过程两种角色。

本小节讲述 BEGIN、END、initialization 和 finalization 背后所隐含的所有代码。你将看到 *System.pas* 与编译器如何协调地管理你的每一个单元、例程与代码块。

4.1 变量的初始化与结束化

部分数据类型的变量是有初始化与结束化过程的。在什么时候调用这些初始化和结束化过程，取决于所定义变量的生命周期。对于局部变量来说，其生命周期自进入例程开始，到结束例程时终止。也就是说，局部变量的初始化与结束化过程，在例程的初始化与结束化过程中发生。

具有初始化与结束化过程的数据类型如表 4-1 所示。

表 4-1　具有初始化与结束化过程的数据类型

	初始化	结束化	备　注
长字符串	内置于_InitializeArray()	内置于_FinalizeArray()	只有在数组或记录中的字符串会通过内置于这两个例程中的代码来完成初始化与结束化
宽字符串			
动态数组			
变体			
接口			
类	TObject.InitInstance()	Tobject.FreeInstance()	
静态数组	_InitializeArray()	_FinalizeArray()	初始化所有具有“引用计数”特征的数组元素
记录	_InitializeRecord()	_FinalizeRecord()	初始化所有具有“引用计数”特征的记录域

4.1.1　初始化的必要性

为什么要初始化？或者更进一步地提出问题：为什么仅有这些类型要被初始化呢？

应该注意到，上表与“表 2-3 发生引用计数的数据类型”和“表 2-4 有‘增加引用’行为的数据类型”有相似之处。事实上，正是“引用计数机制”使得这些数据类型的变量必须被初始化。

以长字符串为例，这样的代码：

```
var
  Str : String;
begin
  Str := 'abc';
end;
```

在执行“Str := 'abc';”时，系统将调用_LStrAsg()来赋值。而_LStrAsg()中，如果“Pointer(Str)<>Nil”，则需要将 Str 的引用计数减 1。这时，如果局部变量 Str 未经初始化就被使用，则 Pointer(Str)就是一个指向随机地址的指针，这将导致_LStrAsg()中的引用计数操作可能访问非法地址。

在这种情况下，必须将变量初值置为 Nil，以使得其他内部例程可以确认该变量并没有在堆上分配内存。

4.1.2 如何初始化

初始化的目的就是将变量初值置为 Nil。

Delphi 中，对于单独定义的长字符串、宽字符串、动态数组、变体以及接口变量，会直接操作栈来完成初始化。关于这一点，可以参见第 3 章中“完全汇编例程与内嵌汇编例程”的有关描述。

记录和数组的初始化如同它们的“增加引用”一样，也是操作其中的一些子域。所使用的“初始化表”与“增加引用”中使用的“引用列表(数组)”是同一个表。

记录的“引用列表”中记录了所有具有引用计数特征的域。数组的引用列表要相对简单一些，只有数组基于长字符串、宽字符串、动态数组、变体以及接口变量定义时，才会有一个记录来保存基类型的信息。

以数组为例，其初始化代码如下：

```
procedure _InitializeArray(p: Pointer; typeInfo: Pointer;
  elemCount: Cardinal);
var
  FT: PFieldTable;
// ...
  case PTypeInfo(typeInfo).Kind of
    tkArray:
      begin
        FT := PFieldTable(Integer(typeInfo) +
          Byte(PTypeInfo(typeInfo).Name[0]));
        while elemCount > 0 do
        begin
          _InitializeArray(P, FT.Fields[0].TypeInfo^, FT.Count);
          Inc(Integer(P), FT.Size);
          Dec(elemCount);
        end;
      end;
// ...
end;
```

这里使用 FT 变量来定义初始化记录的字段表。由于最多只有一个记录，所以 FT.Fields[0].TypeInfo^代表了所有数组元素的数据类型。考虑到多维数组的情况，这里递归调用_InitializeArray()来初始化每一个数组元素。

4.1.3　如何结束化

结束化过程要相对简单得多。局部变量在栈上占用的空间通过栈操作即可释放。记录和数组中的元素可以通过初始化表来结束化。对于不在初始化表中的域或数组元素，在释放记录和数组占用的栈空间的同时就被释放了。

如果变量、域(记录)或元素(数组)具有引用计数特征，那么结束化过程中，都必将调用到相应数据类型中减少引用计数的例程——例如长字符的内部例程_LStrClr()和_LStrArrayClr()。

■ _New()与 Initialize()

如果是全局变量，则记录变量的空间被分配在数据区，这会发生两种情况：

- 操作系统的文件装载程序会自动将 BSS 节初始化为 0，因此无初值的记录变量的初始化可以自动完成；
- 有初值的记录变量不存在初始化的问题。由于它被放在 DATA 节中，因而也不会被自动初始化。

如果记录(而非记录指针)是局部变量，那么会有整个记录大小的空间直接在栈上分配，这时编译器会在例程初始化代码中加入_InitializeRecord()来初始化这个变量。例如：

```
type
  TRec = record
    i : integer;
    s : String;
  end;

procedure InitialiseTest;
var
  r : TRec;
begin
  r.i := 3;
end;
```

这段代码在 CPU 窗体的汇编代码是这样的：

```
Project2.dpr.16: procedure InitialiseTest;
Project2.dpr.17: begin
00407D40 55              push ebp
00407D41 8BEC            mov ebp,esp
```

```
00407D43 83C4F8          add esp,-$08          // 在栈上分配SizeOf(TRec)大小的空间
00407D46 8D45F8          lea eax,[ebp-$08]     // 取变量 r 的地址
00407D49 8B15247D4000    mov edx,[$00407d24] // TypeInfo(TRec)
00407D4F E8D8BCFFFF      call @InitializeRecord // 记录初始化
00407D54 33C0            xor eax,eax           // 以下代码向栈上填入一个异常帧
00407D56 55              push ebp
00407D57 68857D4000      push $00407d85
00407D5C 64FF30          push dword ptr fs:[eax]
00407D5F 648920          mov fs:[eax],esp
Project2.dpr.18: r.i := 3;
// ...
```

编译器在栈上填入异常帧，这使得无论例程 InitializeTest()中的代码运行情况如何，例程结束化时的_FinalizeRecord()调用都必然被执行到。这相当于如下的代码：

```
 InitialiseRecord(r, TypeInfo(TRec));
try
// 1. 以上代码被编译在begin关键字的实现代码中

 // ...
 // 在InitialiseTest()中的代码

// 2. 以下代码被编译在end关键字的实现代码中
finally
  _FinalizeRecord()
end;
```

但如果是记录指针，那么开发人员通常会调用 New()例程来为它分配空间。为此编译器在_New()例程中调用了_Initialize()来初始化：

```
procedure _Initialize(p: Pointer; typeInfo: Pointer);
begin
  _InitializeArray(p, typeInfo, 1);
end;

function _New(size: Longint; typeInfo: Pointer): Pointer;
begin
  GetMem(Result, size);
  if Result <> nil then
    _Initialize(Result, typeInfo);
end;
```

_Initialize()直接调用_InitializeArray()，并把记录当成一个元素的数组传入。而在_InitializeArray()中，如果判定到 typeinfo 是记录，则将再次调用到_InitializeRecord()：

```
procedure _InitializeArray(p: Pointer; typeInfo: Pointer;
  elemCount: Cardinal);
// ...
    tkRecord:
      begin
        FT := PFieldTable(Integer(typeInfo) +
          Byte(PTypeInfo(typeInfo).Name[0]));
        while elemCount > 0 do
        begin
          _InitializeRecord(P, typeInfo);
// ...
```

接下来的代码，看起来像是一个怪圈：_InitializeRecord()将再调用例程_InitializeArray()。_InitializeRecord()的实现代码如下：

```
procedure _InitializeRecord(p: Pointer; typeInfo: Pointer);
var
  FT: PFieldTable;
  I: Cardinal;
begin
  FT := PFieldTable(Integer(typeInfo) + Byte(PTypeInfo(typeInfo).Name[0]));
  for I := FT.Count-1 downto 0 do
    _InitializeArray(Pointer(Cardinal(P) + FT.Fields[I].Offset),
      FT.Fields[I].TypeInfo^, 1);
end;
```

_InitializeRecord()从类型信息记录(FieldTable)中列举每一个使用了引用机制的域，并将每个域作为“一个元素的数组”，再次调用_InitializeArray()来初始化该域的值。

可以这样来说明_InitializeRecord()与_Initialize()之间的差异：_Initialize()中将记录作为“一个元素的数组”传入_InitializeArray()；而在_InitializeRecord()中，将通过 typeInfo 列举记录中每一个需要初始化的域，并将该域(地址)作为“一个元素的数组”传入_InitializeArray()。

4.2　例程的初始化与结束化

例程没有专门用来初始化和结束化的内部函数或过程。

例程初始化时进行的操作包括：

- 生成传值参数的备份，并替换原参数的传入地址；

- 在栈上为局部变量分配空间，并初始化具有引用计数特征的变量；
- 为数组和记录调用初始化过程，初始化其中具有引用计数特征的域(记录)或元素(数组)。

例程初始化过程中不会有堆操作。

例程的结束化时进行的操作包括：

- 清除传值参数的备份；
- 按照调用约定传出 Result；
- 清除局部变量在栈上分配的空间，并为具有引用计数特征的变量调用减少引用计数的内部例程。如果引用计数减至 1 以下，则释放变量在堆上的内存占用；
- 为数组和记录调用结束化过程，以使其中的引用计数特征的域(记录)或元素(数组)，得到与局部变量相类同的结束化操作。

例程的初始化和结束化过程的具体操作，在如下章节中有更细节的描述：

- 第 2 章第 6 节：引用—计数—写复制与类型信息；
- 第 3 章第 4 节：例程入口参数及调用约定；
- 第 3 章第 6 节：完全汇编例程与内嵌汇编例程；
- 第 4 章第 1 节：变量的初始化与结束化。

4.3 单元初始化与结束化

4.3.1 单元初始化与结束化的内部例程

Delphi 会初始化每一个单元，单元结束化则是初始化的逆过程。它们对应于 *System.pas* 中的两个内部例程：

```
procedure InitUnits;
procedure FinalizeUnits;
```

编译器会采用深度优先遍历算法扫描项目的所有单元，然后生成一个初始化表。编译器保证每个单元在初始化表中不会重复，从而使得每一个单元只会被初始化一次。

从执行流程的角度上来说，等到例程 `InitUnits()`被执行时，初始化表已经在内存中准备好了——这些是编译器和操作系统的模块初始化程序所做的工作。

初始化表的结构如下：

```
PackageUnitEntry = packed record
  Init, FInit : Pointer;
end;

UnitEntryTable = array [0..9999999] of PackageUnitEntry;
PUnitEntryTable = ^UnitEntryTable;

PackageInfoTable = packed record
 UnitCount : Integer;   { number of entries in UnitInfo array; always > 0 }
 UnitInfo : PUnitEntryTable;
end;

PackageInfo = ^PackageInfoTable;
```

变量 InitContext 的域 InitTable 指向初始化表。

真正的初始化过程非常简单，例程 InitUnits()核心部分的实现代码如下：

```
procedure InitUnits;
var
  Count, I: Integer;
  Table: PUnitEntryTable;
  P: Pointer;
// ...
   while I < Count do                    // 0..UnitCount-1，列举全部的单元初始化记录
   begin
     P := Table^[I].Init;
     Inc(I);
     InitContext.InitCount := I;         // 记录初始化过的单元数量
     if Assigned(P) then
       TProc(P)();                       // 把初始化代码入口作为一个无参数过程调用
   end;
// ...
end;
```

Table^[I].Init 即是单元文件 initialization 节的代码在内存中的入口地址。同样的，Table^[I].FInit 即是 finalization 节的代码入口地址。在结束化例程 FinalizeUnits()中，直接使用在初始化过程中记录的 InitContext.InitCount 值，列举并调用 Table^[I].FInit。

如果 InitContext.InitCount=0，则调用 procedure FinalizeUnits()时不会有任何实际的操作发生。

单元初始化与结束化分别在模块调入和卸载时发生。如果单元没有 initialization 节和 finalization 节，那么虽然在初始化表中有相应的单元记录，但它的单元入口地址为 Nil。

4.3.2 其他初始化例程

在单元初始化的关键字 initialization 或 begin 之中，在用户代码被执行到之前，编译器通常会填入一些调用初始化例程的代码。在这些例程包括：

```
procedure _InitResStrings;
procedure _InitResStringImports;
procedure _InitImports;
procedure _InitWideStrings;
```

其中，在单元中使用了宽字符串类型的常量或有初值变量的情况下，编译器会在初始化代码之后填入代码来调用 InitWideStrings()例程。它的作用是将放在 CODE 节中的宽字符串复制到堆，并将它在堆中的地址赋给源代码中声明它的变量或者常量。由于宽字符串与长字符串的存储方式和行为是一致的，因此可以参见第 6 章中“数据节与代码节(CODE)”。

宽字符串没有使用引用计数机制，因此它不能像长字符串那样在第一次赋值引用时才检测字符串的有效性并决定是否赋值。所以使用宽字符串类型的常量或有初值变量时，总是要导致单元初始化时调用例程_InitWideStrings()，而使用长字符串时却不必。

其他三个例程的相关细节，请参见第 10 章中“资源字符串”。

4.4 模块初始化与结束化

4.4.1 模块入口代码

编译器为每个 Delphi 模块(可执行程序、动态链接库或包)生成模块入口代码。同类型的程序或模块，其入口代码是相同的。

编译器为可执行程序(*.EXE*)生成的入口代码为：

```
Project1.dpr.9: begin
0044CA98 55              push ebp
0044CA99 8BEC            mov ebp,esp
0044CA9B 83C4F0          add esp,-$10
0044CA9E B8B8C84400      mov eax,$0044c8b8   // 传入单元初始化表地址
0044CAA3 E8D006FEFF      call @InitExe       // 调用模块初始化例程
```

为动态链接库(.*DLL*)生成的入口代码为：

```
Project2.dpr.19: begin
00B81FD4 55              push ebp
00B81FD5 8BEC            mov ebp,esp
00B81FD7 83C4C4          add esp,-$3c
00B81FDA B85C1FB800      mov eax,$00b81f5c   // 传入单元初始化表地址
00B81FDF E8F435FFFF      call @InitLib       // 调用模块初始化例程
```

为包(.*BPL*)生成的入口代码为：

```
Package1.dpk.33: end.
// 跳转到模块初始化例程，是一个相对寻址的短跳转
04A517B0 E9C3FEFFFF      jmp @InitPkg
```

除了包之外，入口代码都主要负责传入单元初始化表的地址。其后的所有初始化工作，全部交由 *System.pas* 中的模块初始化例程完成。这些例程分别是：

```
// define in SysInit.pas
procedure _InitLib;
procedure _InitExe(InitTable: Pointer);
Function  _InitPkg(Hinst: Integer; Reason: Integer;
  Resvd: Pointer): LongBool; stdcall;
```

4.4.2　编译器决定的程序执行流程

编译器总是按照固定的流程和代码来生成项目的二进制文件。例如一个简单的项目：

```
program GuiProg;
uses
  Dialogs;
begin
  ShowMessage('Test');
end.
```

它编译的代码为：

```
GuiProg.dpr.4: begin
00451F5C 55              push ebp
00451F5D 8BEC            mov ebp,esp
00451F5F 83C4F0          add esp,-$10
00451F62 B8841D4500      mov eax,$00451d84
00451F67 E8F43CFBFF      call @InitExe
GuiProg.dpr.5: ShowMessage('Test');
00451F6C B8841F4500      mov eax,$00451f84
00451F71 E8D2F8FFFF      call ShowMessage
```

```
GuiProg.dpr.6: end.
00451F76 E8E11DFBFF      call @Halt0
```

上面的汇编代码是顺序执行的，编译器只是在这个流程的结尾加入了一行“call @Halt0”。Delphi 并没有刻意为模块准备结束化代码或者指定结束化代码的地址，它只须按照这样的方式来编译即可。

对于可执行程序(.*EXE*)来说，用户在项目文件(.*DPR*)的 begin..end 之间加入的代码决定了其所有的功能，而编译器只是把.*DPR* 文件按照.*EXE* 方式生成，并将用户编写的代码编译成二进制代码，放到入口代码和结束时的“call @Halt0”之间。

其他类型的文件与可执行程序并不一致。在下一章“面向 Windows 开发的基本实现”中，将详细地讲述所有的模块初始化与结束化例程，以及它们不同执行流程。

第 5 章

面向 Windows 开发的基本实现

Delphi 可以实现 Windows 系统中的大多数可执行文件类型，这主要包括可执行程序(.*EXE*)和动态链接库(.*DLL*)。其他的一些文件类型，通常以动态链接库的形式编译，只是输出的文件扩展名不同而已(例如.*CPL*，.*SCR*，.*OCX* 等)。

Delphi 使用动态链接库的格式实现了自己的库文件：包(.*BPL*)。

本章全面解析 Delphi 支持这些文件格式的每一个细节。

5.1 Win32 应用程序：EXE

5.1.1 适应 Win32 应用程序的最简化内核

Nico Bendlin 的 MiniDExe 很好地演示了不使用任何 Delphi 例程来实现一个 Win32 应用程序的方法。对于一个可执行程序.*EXE* 来说，只须满足如下条件，就可以在被 Windows 系统中执行：

- 是一个以.*EXE* 方式生成的格式正确的 PE (Portable Executable)文件
- 有一个正确的入口地址并记录在 PE 格式文件的头部

编译器会按这样的规则生成文件模块，并将一段入口代码的地址记录在 PE 格式文件的头部。这段入口代码固定地调用 *System.pas* 中的例程_InitExe()。

因此，可以进一步简化 Nico Bendlin 的 MiniDExe：

```
//系统初始化单元
unit SysInit;
```

```
interface

var
  TlsIndex: Integer = -1;
  TlsLast: Byte;

const
  PtrToNil: Pointer = nil;

implementation

end.

//系统内核单元
unit System;

interface

procedure _InitExe;
procedure _HandleFinally;
procedure _Halt0;

const
  Kernel32 = 'kernel32.dll';
  User32 = 'user32.dll';

type
  TGUID = record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;

implementation

procedure ExitProcess (uExitCode: LongWord); stdcall;
  external kernel32 name 'ExitProcess';

procedure _InitExe;
asm
end;

procedure _HandleFinally;
asm
```

Delphi 源代码分析

```
end;

procedure _Halt0;
begin
  ExitProcess(0);
end;

end.

//示例程序，目标文件为3584 Bytes
program MiniDExe;

function ShowMessageBox(hWnd: LongWord; lpText, lpCaption: PChar;
  uType: LongWord): Integer;
  stdcall; external user32 name 'MessageBoxA';

const
  MB_ICONINFORMATION = $00000040;

begin
  ShowMessageBox(0, 'Written in pure Delphi!', 'Hello World!',
    MB_ICONINFORMATION);
end.
```

通过导入外部例程，上面的代码可以调用全部 Win32API 以及其他的动态链接库资源。可能存在的问题包括：

- 由于没有处理在入口代码中传入的单元初始化表，因此单元文件的初始化和结束化节是无效的；
- 由于没有内部例程的支持，因此一些兼容类型的赋值不能进行。例如长字符串与宽字符串的赋值；
- 由于没有处理_HandleFinally()，因此 try-finally-end 语法是无效的(try-except-end 是由 *SysUtils.pas* 来处理的)。

5.1.2 初始化例程_InitExe()

```
// code from SysInit.pas
procedure _InitExe(InitTable: Pointer);
begin
  TlsIndex := 0;
  HInstance := GetModuleHandle(nil);
  Module.Instance := HInstance;
  Module.CodeInstance := 0;
```

```
  Module.DataInstance := 0;
  InitializeModule;
  _StartExe(InitTable, @Module);
end;
```

位于 *SysInit.pas* 中的初始化例程_InitExe()其实只完成如下极少的功能：

- 初始化系统全局变量 TlsIndex 和 HInstance；
- 初始化模块信息记录 Module；
- 调用 InitializeModule()来初始化内部模块表；
- 调用_StartExe()。

其中模块信息记录 Module 是非常重要的一个系统内部变量，其类型定义如下：

```
PLibModule = ^TLibModule;
TLibModule = record
  Next: PLibModule;
  Instance: LongWord;              // 模块的实例句柄
  CodeInstance: LongWord;          // 模块的代码实例句柄
  DataInstance: LongWord;          // 模块的数据实例句柄
  ResInstance: LongWord;           // 模块的资源实例句柄
  Reserved: Integer;
end;
```

5.1.3　内部模块表管理例程

Delphi 提供了一组用于管理模块信息记录和内部模块表的例程：

```
procedure RegisterModule(LibModule: PLibModule);
procedure UnregisterModule(LibModule: PLibModule);
function FindHInstance(Address: Pointer): LongWord;
function FindClassHInstance(ClassType: TClass): LongWord;
function FindResourceHInstance(Instance: LongWord): LongWord;
function LoadResourceModule(ModuleName: PChar;
  CheckOwner: Boolean = True): LongWord;
procedure EnumModules(Func: TEnumModuleFunc; Data: Pointer); overload;
procedure EnumResourceModules(Func: TEnumModuleFunc;
  Data: Pointer); overload;
procedure EnumModules(Func: TEnumModuleFuncLW; Data: Pointer); overload;
procedure EnumResourceModules(Func: TEnumModuleFuncLW;
  Data: Pointer); overload;
procedure AddModuleUnloadProc(Proc: TModuleUnloadProc); overload;
procedure RemoveModuleUnloadProc(Proc: TModuleUnloadProc); overload;
procedure AddModuleUnloadProc(Proc: TModuleUnloadProcLW); overload;
```

```
procedure RemoveModuleUnloadProc(Proc: TModuleUnloadProcLW); overload;

// define in SysInit.pas
procedure InitializeModule;
procedure UninitializeModule;
```

需要注意的是，这个内部模块表与操作系统的模块表并不一致。Delphi 的内部模块表只记录当前模块加载的包，而操作系统的模块表记录当前进程加载的所有模块。

初始情况下，*.EXE* 的内部模块表只有一个记录，即当前模块(*.EXE*)的信息记录。

SysInit 单元的例程 InitializeModule()只是简单地调用 *System* 单元中的 RegisterModule()。例程 RegisterModule()则负责把模块放在系统模块列表的头部。

5.1.4 .EXE 启动例程_StartExe()

_InitExe()将由启动代码传入的单元初始化表，以及定义在 *SysInit.pas* 中的当前模块信息指针传给_StartExe()，从而真正地启动.*EXE* 的初始化过程。

```
procedure _StartExe(InitTable: PackageInfo; Module: PLibModule);
begin
  RaiseExceptionProc := @RaiseException;     // 初始化异常引发程序的指针
  RTLUnwindProc := @RTLUnwind;               // 初始化异常展开程序的指针

  InitContext.InitTable := InitTable;
  InitContext.InitCount := 0;   // 在InitUnits()中将使用InitCount对初始化过
                                // 的单元进行计数
  InitContext.Module := Module;             // .EXE的模块信息
  MainInstance := Module.Instance;          // 模块句柄

  IsLibrary := False;
  InitUnits;
end;
```

在_StartExe()中，主要处理初始化上下文(InitContext)。这个系统内部变量用于记录初始化和结束化中的一些重要信息。结构如下：

```
type
  PInitContext = ^TInitContext;
  TInitContext = record
    OuterContext:    PInitContext;        { 当前上下文的备份      }
    InitTable:       PackageInfo;         { 单元初始化信息表      }
    InitCount:       Integer;             { InitTable的长度       }
    Module:          PLibModule;          { 当前模块信息指针      }
```

```
  DLLSaveEBP:      Pointer;          { saved regs for DLLs }
  DLLSaveEBX:      Pointer;          { saved regs for DLLs }
  DLLSaveESI:      Pointer;          { saved regs for DLLs }
  DLLSaveEDI:      Pointer;          { saved regs for DLLs }
  ExitProcessTLS:  procedure;        { 进程TLS退出例程      }
  DLLInitState:    Byte;             { 0 = package, 1 = DLL shutdown, 2 = DLL
    startup }
end platform;
```

该记录中各个域的使用情况如表 5-1。

表 5-1　各个域的使用情况

域	.EXE	.BPL	.DLL
OuterContext	①		
InitTable	初始化，结束化	初始化，结束化	初始化，结束化
InitCount	初始化，结束化	初始化，结束化	初始化，结束化
Module	初始化，结束化	初始化，结束化	初始化，结束化
DLLSaveEBP			初始化，结束化
DLLSaveEBX			初始化，结束化
DLLSaveESI			初始化，结束化
DLLSaveEDI			初始化，结束化
ExitProcessTLS	初始化，结束化	初始化，结束化	初始化，结束化
DLLInitState	0	0	1..4

5.1.5　应用程序的结束化控制

在 Delphi 中，有四种情况可以导致一个应用程序(进程)的结束：

- 程序代码执行完成，执行.*DPR* 的结束语句“*END.*”正常退出；
- 应用程序内部调用例程 procedure halt()；
- 应用程序内部调用操作系统 API：procedure ExitProcess()；
- 应用程序内部或者其他应用程序调用操作系统 API：function TerminateProcess()。

① InitContext 中 OuterContext 域的使用相对复杂，在本章第 5 节“其他”中统一描述。

最后两种方法都是用操作系统 API 来使进程中止的，这种情况下，Delphi 不做任何的处理。

前两种方法最终都将调用内部例程_Halt0()。在这个例程中，与.*EXE* 相关的代码有：

```
procedure _Halt0;
var
  P: procedure;
begin
  //检查并执行退出过程
  if InitContext.DLLInitState = 0 then // ,exe module's DLLInitState = 0
    while ExitProc <> nil do
    begin
      @P := ExitProc;
      ExitProc := nil;
      P;
    end;

  //检查并显示系统错误信息，WriteErrorMessage()例程将自动识别是否是控制台输出
  if ErrorAddr <> nil then
  begin
    MakeErrorMessage;
    WriteErrorMessage;
    ErrorAddr := nil;
  end;

  //检查模块的初始化上下文
  while True do
  begin
    // ...

    //单元结束化
    FinalizeUnits;

    //执行与procedure UninitializeModule()例程相同的操作
    if (InitContext.DLLInitState <= 1) or (ExitCode <> 0) then
    begin
      if InitContext.Module <> nil then
        with InitContext do
        begin
          //从内部模块表中卸载，但不并表明模块从内存中卸载
          UnregisterModule(Module);
          //从内存中卸载模块载入的资源模块
          if (Module.ResInstance <> Module.Instance) and
            (Module.ResInstance <> 0) then
            FreeLibrary(Module.ResInstance);
```

```
      end;
    end;

    // ...

    //如果当前模块是.EXE，则试图执行ExitProcessProc()，然后退出进程
    if InitContext.OuterContext = nil then
    begin
      if Assigned(ExitProcessProc) then
        ExitProcessProc;
      ExitProcess(ExitCode);
    end;

    InitContext := InitContext.OuterContext^
  end;
end;
```

在退出过程 ExitProc 的检查时，没有使用 IF 来检测 "ExitProc<>Nil"，而是采用了 while 循环，这使得可以在 ExitProc 中再次给 ExitProc 赋值，从而形成 ExitProc 的链表。这意味着类似下面的代码能被正常地执行：

```
var
  OldExitProc : Pointer;

procedure NewExitProc;
begin
  showmessage('test');
  ExitProc := OldExitProc;    // 在退出过程中重设ExitProc
end;

procedure ReplaceExitProc;
begin
  OldExitProc := ExitProc;
  ExitProc := @NewExitProc;
end;
```

应当使用 *SysUtils.pas* 中的例程 AddExitProc() 来安全地操作 ExitProc。

在 *System.pas* 单元的内部，还隐藏着一个模块退出过程列表。这个列表中的退出过程，是通过在 procedure UnregisterModule() 中调用例程 NotifyModuleUnload() 来执行的。可以使用例程 AddModuleUnloadProc() 和 RemoveModuleUnloadProc() 来维护模块的退出过程列表。

值得注意的是 Halt0() 中还有一个 ExitProcessProc 的检查过程。在 Delphi 看来：

ExitProc 是表明当前可执行模块(.*EXE*)的退出过程；ModuleUnloadProc 是任意类型模块(.*EXE*、.*DLL*、.*BPL*)的退出过程；而 ExitProcessProc 则是当前进程的退出过程。对于.*EXE* 来说，当前模块与当前进程有非常紧密的关系，但 ExitProcessProc 与 ExitProc 却完全无关。

因此，只有.*EXE*会使用到ExitProc 和ExitProcessProc过程。而ModuleUnloadProc 是当前进程中所有模块都可以使用的。如果动态链接库和包不是使用“带运行期库”方式编译的，那么，ExitProc 和 ExitProcessProc 对于这两种模块来说是无意义的。

只有条件“InitContext.OuterContext = nil”为真，才表明当前模块是.*EXE* 模块。这种情况下，_Halt0()才执行操作系统 API：ExitProcess()退出进程。

Halt0()例程的结束化处理中也包括单元结束化。不过需要注意的是：系统是在模块结束化之前调用例程 FinalizeUnits()。这意味着在 ModuleUnloadProc 和 ExitProcessProc 中无法使用某些类型的全局变量(例如对象)，但是在 ExitProc 中，可以使用所有的东西。

5.2 32 位的 DOS：控制台应用程序

5.2.1 控制台应用程序的模块入口代码

控制台应用程序的模块入口代码较普通的.*EXE* 略有不同：

```
Project2.dpr.8: begin
00408228 55              push ebp
00408229 8BEC            mov ebp,esp
0040822B 83C4F0          add esp,-$10
0040822E A1B0934000      mov eax,[$004093b0]        // Pointer of IsConsole
00408233 C60001          mov byte ptr [eax],$01     // IsConsole := True
00408236 B8F8814000      mov eax,$004081f8
0040823B E844C7FFFF      call @InitExe
```

这里加入了设置系统全局变量 IsConsole 的代码。如果有下列条件之一，则编译器生成上述的入口代码，使内部例程_InitExe()被调用之前 IsConsole 为真：

- 源代码中加入编译条件{$APPTYPE CONSOLE}；
- 编译命令行上加入-cc 参数；
- 在 IDE 的项目配置中，选中 *Generate console application* 选项。

程序所在的控制台窗口，在模块入口代码被执行之前，就已经被操作系统打开了。更详细地讲：

- 如果父进程有自己的控制台，程序就继承这个控制台；
- 否则，操作系统为程序创建一个控制台窗口。

Delphi 的内核只是使用这个窗口而已，并不负责它的开启。操作系统判断对于一个.*EXE*程序是否需要开启控制台窗口的依据，是该程序模块的 *PE* 头结构中是否指定了要使用 *CONSOLE* 系统。这一点，在下一章的“Delphi 的积木艺术(PE)”中详细地讲述。

除了在入口代码和 *PE* 头结构中进行的细微差异，一个控制台应用程序与 GUI 模式的.*EXE* 并没有不同；它们的初始化和结束化是完全一致的。在 Delphi 内核中，只有 `procedure WriteErrorMessage()`这一例程需要检测 `IsConsole` 变量，以决定当系统出错时，是在控制台上显示错误，还是弹出错误对话框。

5.2.2　控制台应用程序的最小化实现

编译器与操作系统为控制台应用程序的执行准备好了一切，接下来只须接管控制台的输入输出即可。这包括：

- 使用操作系统 API：`GetStdHandle()`获取控制台的句柄；
- 使用操作系统 API：`ReadFile()`和`WriteFile()`操作控制台的输入输出。

下面的示例演示了控制台输出：

```
program MiniConsole;
{$APPTYPE CONSOLE}

function WriteFile(hFile: Integer; const Buffer;
  nNumberOfBytesToWrite: Cardinal;
  var lpNumberOfBytesWritten: Cardinal;
  lpOverlapped: Pointer): Integer; stdcall;
  external kernel32 name 'WriteFile';

function GetStdHandle(nStdHandle: Integer): Integer; stdcall;
  external kernel32 name 'GetStdHandle';

const
  STD_OUTPUT_HANDLE = Integer(-11);

var
  s : ShortString = 'This is a Console Application.';
  HStdOut : integer;
  Dummy: Cardinal;
```

Delphi 源代码分析

```
begin
  Dummy := Byte(S[0]);
  HStdOut := GetStdHandle(STD_OUTPUT_HANDLE);
  WriteFile(hStdOut, S[1], Dummy, Dummy, nil);
end.
```

要使用最小化内核编译这个程序，只须在 *System.pas* 中加入全局变量“IsConsole”的定义即可。

5.2.3 控制台应用程序的 Delphi 实现

下面的应用能够在最小化内核中编译出可执行目标代码：

```
program TestMini;
const
  MaxAsc  = 26;
var
  i : Integer;
  s : String[MaxAsc];
begin
  s[0] := chr(MaxAsc);
  for i := 1 to MaxAsc do
    byte(s[i]) := ord('a') -1 + i;

  // Writeln(s);
end.
```

其中，ord()和 chr()两个函数是编译器内置的，而 byte()是强制类型转换。因此，系统并不需要请求更多的代码(通常在标准 Pascal 中声明过的函数都被编译器内置)。但是，如果要使用 Writeln()函数向控制台输出，那么要同时在最小化内核中添加的内容包括：

```
function _WriteBytes(var t: TTextRec; const b; cnt : Longint): Pointer;
function _WriteSpaces(var t: TTextRec; cnt: Longint): Pointer;
function _WriteString(var t: TTextRec; const s: ShortString;
  width: Longint): Pointer;
function _WriteOString(var t: TTextRec; const s: ShortString): Pointer;
function _WriteLn(var t: TTextRec): Pointer;
function _RewritText(var t: TTextRec): Integer;
function _Close(var t: TTextRec): Integer;
function _Flush(var t: TTextRec): Integer;
procedure __IOTest;

function OpenText(var t: TTextRec; Mode: Word): Integer;
function TryOpenForOutput(var t: TTextRec): Boolean;
```

```
procedure SetInOutRes(NewValue: Integer);
```

其中包括文本文件操作、IO 检查以及与 Writeln()相关的一些内部函数。此外，一些相关的系统常量与变量也需要包括进来。为了使 Writeln()一行代码得以编译通过，大概需要在 *System.pas* 中加入 22K 的源代码！如果需要使用 Readln()等例程来操作控制台，那么需要添加的函数将会更多。

在 *System.pas* 中与控制台相关的代码主要包括：

- 将操作系统提供的控制台句柄以文本文件的形式描述为基本输入、输出系统；
- 文本文件操作例程；
- 不同入口参数的 Writeln()和 Readln()例程；
- IO 测试。

5.2.4　文件操作例程与控制台应用程序

在 Delphi 中，至少有三种方法可以操作文件，如下所示：

- 定义 File 文件类型变量，使用 Pascal 基本例程操作文件；
- 定义文件句柄变量，使用 Win32 API 操作文件；
- 定义流对象变量，使用基于流的方式操作文件。

Delphi 在 *System.pas* 中实现了文件类型(File)以及文本文件类型(Text)，一方面使得 Delphi 能够更全面地兼容 Pascal 的语言特征，另一方面也使得在开发人员看来，控制台应用的接口与 DOS 程序有更多的相似之处。

在这样的相似性背后，需要明确的是：

- 控制台是一个真实的窗口应用程序，操作系统给控制台提供的资源(API、动态链接库、CPU 模式以及内存等)与其他的图形界面(GUI)应用程序并没有本质的不同。
- 控制台应用程序是一个真正的 Win32 程序[1]，它只是在虚拟 DOS 的窗口中运行而已。除了界面上的相似，该窗口不具有 DOS 操作系统的任何特征。
- 任何进程都可以开启自己的控制台窗口。这并不需要调用 *CMD.EXE* 来实现，操作系统提供了足够的 API 使程序开启并控制自己的控制台。

(1)　此资料中提供的信息并不确切，控制台程序并非运行在 V86 模式，也无须用 *Winoldap.mod* 来载入。*Winoldap.mod* 用于载入一些早期在 DOS 模式下开发的 16 位应用程序(在 WinNT 系统中 Win16 位程序和 DOS 程序都使用 *Ntvdm.exe* 载入)，而控制台程序是 32 位的 Windows 程序。

Delphi 使用 Text 文件类型隐藏了这一切细节，使得控制台程序更像一个 DOS BOX，而 Delphi 做的，只不过是 Writeln()或者 Readln()。

这一切只是表象而已。

Text 文件类型实际上是通过 TTextRec 类型记录来实现的。这个记录定义了四个重要的域：

```
{* must match the size the compiler generates: 460 bytes *}
TTextRec = packed record
  // ...
  OpenFunc: Pointer;
  InOutFunc: Pointer;
  FlushFunc: Pointer;
  CloseFunc: Pointer;
  // ...
end;
```

这些指针指明了该文件使用哪些内部例程来操作。对于 Text 文件记录来说，这些域都是开放的，因此可以修改这些域，使用其他方式读写文本文件——如果能确定这些新的例程的安全性的话。

编译器使文本文件打开例程指向下面三个函数之一，进而调用到例程 OpenText()：

```
//procedure Reset(var F [: File; RecSize: Word ] );
function _ResetText(var t: TTextRec): Integer;
begin
  Result := OpenText(t, fmInput);
end;

// procedure Rewrite(var F: File [; RecSize: Word ] );
function _RewritText(var t: TTextRec): Integer;
begin
  Result := OpenText(t, fmOutput);
end;

// procedure Append(var F: Text);
function _Append(var t: TTextRec): Integer;
begin
  Result := OpenText(t, fmInOut);
end;
```

OpenText()初始化文本文件记录的域 OpenFunc，使其指向内部例程 TextOpen()。然后调用之。在 TextOpen()例程中，则进一步地记录其他域，使其余三个指针分别指向内部例程：

```
// InOutFunc读/写请求，指向如下例程之一
function TextIn(var t: TTextRec): Integer;
function TextOut(var t: TTextRec): Integer;

// 对控制台，FlushFunc指向例程FileNOPProc()；对普通文件指向TextOut()
function FileNOPProc(var t): Integer;
function TextOut(var t: TTextRec): Integer;

// CloseFunc指向例程TextClose()
function TextClose(var t: TTextRec): Integer;
```

函数 TextOpen()会根据文本文件记录的 Name 和 Mode 域来判定打开什么文件以及用什么方式打开。同时，也据此判定是操作物理文件还是控制台。其具体的规则如表 5-2 所示。

表 5-2

Name	Mode	操　作	备　注
空	(fmInput and 3) = 1	控制台Input	
空	(fmOutput and 3) = 2	控制台Output	如果记录等于变量ErrOuput，则是错误输出控制台，否则是输出控制台
非空	(fmInput and 3) = 1	文件开启，只读	Reset(F), OpenMode := GENERIC_READ
非空	2	文件开启，写	Rewrite(F), OpenMode := GENERIC_WRITE
非空	(fmInOut and 3) = 3	文件开启，读写	Append(F), OpenMode := GENERIC_READ OR GENERIC_WRITE

如果操作对象是文件，则以 OpenMode 参数调用 API：CreateFileA()来开启文件并将文件句柄填入 Handle 域[①]；如果操作对象是控制台，则调用 API：GetStdHandle()来取得控制台的句柄并填入 Handle 域。

接下来，控制台与文本文件的操作就完全一致了：TextIn()和 TextOut()例程只是调用 API：ReadFile()和 WriteFile()而已，并不区分操作对象是控制台还是文本文件。

至于使用 Readln()与 Writeln()来操作控制台或文件，则不过是 Delphi 用 compiler magic 以及字符串处理的一些技巧而已，与控制台的实现机制已经无关了。

① 需要了解的是，一旦文本文件成功打开，则文件已经被读入 128 Bytes 到缓冲区(缺省情况下文本文件记录的 Buffer 大小)，如果文件长度小于 128 字节，则文件的 Eof 标志置为真，对于控制台来说，这一切都是无意义的。

5.2.5 控制台的开启与关闭

Delphi 为控制台应用程序所做的最后一件事，仅仅是定义了三个变量，并在 *System.pas* 单元初始化和结束化的时候，负责开启与关闭它而已。

```
Input: Text;          { Standard input          }
Output: Text;         { Standard output         }
ErrOutput: Text;      { Standard error output   }
```

Delphi 在初始化时并不强行初始化控制台系统，它只是将上面三个变量的 Mode 域置为 fmClosed。在 Write() 和 Read() 例程的内部(准确地说，是_WriteBytes()和_ReadChar()例程的内部)，总是会尝试打开一个关闭的文件记录；如果记录是控制台，则试图调用打开文本文件的例程；如果是文件，则返回运行期错误。

从这里可以看到，虽然 Delphi 为每个模块都初始化了控制台变量，但是直到调用 Write() 或 Read()，或者在显式调用 Reset(Input)和 Rewrite(Output)来打开控制台之前，这些变量的 Mode 都是 fmClosed。也就是说，即使变量 IsConsole 为 True，操作系统也为该应用创建了控制台窗口，但是 Delphi 中的这几个控制台全局变量却仍然是无效的。下面的例子证明了这一点：

```
program LostConsole;
{$APPTYPE CONSOLE}
uses
  Dialogs, SysUtils;
begin
  ShowMessage('IsConsole : ' + BoolToStr (IsConsole, True));

  // 显示为fmClosed
  ShowMessage('INPUT.Mode : $' + IntToHex(TTextRec(INPUT).mode, 4));

  // 显示为无效句柄
  ShowMessage('INPUT.Handle : $' + IntToHex(TTextRec(INPUT).Handle, 4));
end.
```

Delphi 这样做，可以避免在 *.DLL*、*.BPL* 和 GUI 应用程序中编译进 OpenText() 及相关例程的代码。基于同样的原因，内核中退出时显示出错信息的例程是这样写的：

```
procedure WriteErrorMessage;
var
  Dummy: Cardinal;
begin
  if IsConsole then
  begin
```

```
    With TTextRec(Output) do
    begin
      if (Mode = fmOutput) and (BufPos > 0) then
        TTextIOFunc(InOutFunc)(TTextRec(Output));  // flush out text buffer
    end;
    WriteFile(GetStdHandle(STD_OUTPUT_HANDLE), runErrMsg,
      SizeOf(runErrMsg), Dummy, nil);
    WriteFile(GetStdHandle(STD_OUTPUT_HANDLE), sLineBreak, 2, Dummy, nil);
  end
  else if not NoErrMsg then
    MessageBox(0, runErrMsg, errCaption, 0);
end;
```

这里没有直接使用 Writeln()，而是使用 GetStdHandle()例程来取得控制台句柄并输出，从而使内核不必总是编译进 Writeln()及相关的代码。这些措施，可以使目标文件内核减小大约 2K。

5.2.6　CRT 单元与 Input、Output 的重载

Delphi 初始时 Input 和 Output 变量是无效的，对普通开发者来说是一件微不足道的事。毕竟绝大多数开发者都直接使用 Writeln()和 Readln()。但是，这给 CRT 单元的开发者带来了意想不到的收获。

CRT 单元由来已久，甚至在 Borland Pascal for Windows 以及 Delphi 1.0 的时代都还有 WinCRT 单元来模拟它。这个单元在 Turbo Pascal 中用于操作 DOS 屏幕上的光标位置、字符颜色以及键盘事件等。在 DOS 时代，还有大量基于 CRT 单元的第三方开发包，用来实现菜单、对话框等字符模式的用户界面。

从 Delphi 2.0 开始，就再也没有官方的 CRT 单元了。不过，仍然可以找到一些好用的替代品。例如：

- CRT32：Attila Szomor 基于 Frank Zimmer 的 *CRT32.TXT* 编写。
- Crt replacement for Delphi 1.20：Will DeWitt Jr.编写，以下简称 CrtRep。

CRT32 编写的时间较早，除实现了 CRT 的全部功能之外，还加入了鼠标支持等功能，但核心部分有些不稳定。而 CrtRep 是新近才发布在 Borland Code Central 上的，对文本模式支持得非常好，而且代码流畅。

以后者为例。由于 Input 和 Output 需要在例程 Write()和 Read()被调用时才进行初始化。而且被初始化的只是一个入口，即 TTextRec.OpenFunc 指针，其他的操作例程指针，是由 OpenFunc 负责设置的。因此，CrtRep 实现时，把自己的_CrtOpen()例程指针填入

OpenFunc 域，剩下的工作，就全部由_CrtOpen()接管了。

_CrtOpen()中，CrtRep 按照如下的规则替换了 Output 记录的其他三个域(Input 类同)：

```
f.InOutFunc := @_CrtOut;
f.FlushFunc := @_CrtOut;
f.CloseFunc := @_CrtOutClose;
```

这些例程则按照 DOS 的 CRT 单元的规则处理屏幕输出。其中，输出例程_CrtOut()是使用 API：WriteConsoleOutput()来实现输出的，而不是 *System.pas* 中使用的 API：WriteFile()，因此才使 CrtRep 拥有了更强的操作控制台窗体的能力。

Delphi 缺省 Input 和 Output 无效，也给 CrtRep 带来一些小的麻烦。例如 CrtRep 不得不在单元初始化时，显式地调用 Reset(Input)和 Rewrite(Output)来使这两个变量有效。因为在普通的 Delphi 控制台程序中，可能发生的操作通常就只是 Write()或 Read()，而在使用 CRT 单元的控制台程序中，大多会以类似这样的方法来输出：

```
begin
  ClrScr;                 // 清屏
  GotoXY(10, 30);         // 将光标定位到(10,30)位置
  Writeln('测试CRT单元.');
end.
```

若没有预先使 Input 和 Output 有效，那么 Writeln()前的两行代码不会产生任何效果。

5.3 动态链接库：DLL

5.3.1 丢失的 DllMain()

使用过 C 语言或者汇编语言的 DLL 开发者都清楚地知道：DLL 应该从一个名为 DllMain()的例程开始编写，而编译器也会使 DllMain()成为 DLL 载入时的入口。以 Delphi 方式描述的例程声明如下：

```
function DllMain(hinstDLL: DWORD; fdwReason: DWORD;
  lpvReserved: Pointer): BOOL; stdcall;
```

操作系统在四种情况下，以不同的 fdwReason 参数值调用_DllMain()入口。具体如下：

- DLL_PROCESS_ATTACH：进程试图装入 DLL；
- DLL_PROCESS_DETACH：进程试图卸载 DLL；
- DLL_THREAD_ATTACH：进程开启了一个线程；

- DLL_THREAD_DETACH：进程中的一个线程结束。

■ DLL_PROCESS_ATTACH 与 DLL_PROCESS_DETACH

同一个 DLL 在一个进程中加载时，必将发生、也只发生一次 DLL_PROCESS_ATTACH 调用。这与在进程的哪一个线程中装入是无关的。DLL 卸载时的处理与此类同。

进程中止(TerminateProcess)时，DLL 不会得到任何消息。

■ DLL_THREAD_ATTACH 与 DLL_THREAD_DETACH

操作系统不会为进程的主线程发送以 DLL_THREAD_ATTACH 或 DLL_THREAD_DETACH 为参数的 DllMain()入口调用。

如果装载 DLL 时，进程已经拥有了几个线程，那么操作系统也不会发送与这些已有线程相关的 DLL_THREAD_ATTACH 调用。但是在 DLL 卸载时，可能会得到与它们相关的 DLL_THREAD_DETACH 调用。

当发生 DLL_THREAD_DETACH 调用时，相关的线程仍然存在。因此，动态链接库甚至可以向线程发送消息。但是不应当使用 PostMessage，因为线程可能在此消息被处理之前就已经结束了。

在 Delphi 中，编译器隐含了有关 DllMain()的一切。能够知道的，仅仅是编译器生成了一段入口代码，并调用系统内部例程_InitLib()。这段入口代码是：

```
Project2.dpr,19: begin
00881FD4 55                  push ebp
00881FD5 8BEC                mov ebp,esp
00881FD7 83C4C4              add esp,-$3c
00881FDA B85C1F5800          mov eax,$00581F5c   // 传入单元初始化表地址
00881FDF E8F435FFFF          call @InitLib       // 调用模块初始化例程
```

使用这样的入口代码，有三个目的：

- 在栈顶开辟大小为$3c 个字节的空间；
- 在 eax 寄存器中填入单元初始化表的地址；
- 使用类似于 DllMain()的接口方式调用_InitLib()。

所以，在_InitLib()中可以看到这样的一段注释：

```
// code from SysInit.pas
```

```
procedure _InitLib;
asm
        { ->    EAX InitTable  }
        {       [EBP+8] Hinst  }
        {       [EBP+12] Reason}
        {       [EBP+16] Resvd }
// ...
```

因此，Delphi 中的_InitLib()也就扮演着与DllMain()类似的角色：DLL 的入口点(Entry Point)。

5.3.2 _InitLib()例程

在_InitLib()中，Delphi 仅完成四个操作：

- 如果不是 DLL_PROCESS_ATTACH，则直接转入 4，否则：
 1. 初始化 ModuleIsLib 为 True
 2. 从[EBP+8]位置取出 *DLL* 的模块实例句柄并填入全局变量 HInstance。
 3. 初始化模块信息记录 *Module*，并调用例程 procedure InitializeModule()。
- 对于所有入口的情况，均：
 4. 传入 DllProc 和 TlsProc 参数，并调用例程_StartLib()。

由此可见，_InitLib()的实质是在进程装载时，完成模块在当前进程中的初始化；并负责将每次(线程和进程装载时)的入口参数传给_StartLib()。

5.3.3 _StartLib()例程

为了维护堆栈的完整性，_InitLib()采用了非常不规范的方法调用内部例程_StartLib()。它具体的参数传入情况为：

```
procedure       _StartLib;
asm
        { -> EAX InitTable      }
        {    EDX Module         }
        {    ECX InitTLS        }
        {    [ESP+4] DllProc    }
        {    [EBP+8] Hinst      }
        {    [EBP+12] Reason    }
        {    [EBP+16] Resvd     }
// ...
```

由 EAX 和 EDX 两个参数传入模块的单元初始化表(*InitTable*)和模块信息记录(*Module*)，随

后它们将被填入初始化上下文记录(InitContext)。此外_StartLib()完成如下工作：

- 备份后续代码中要修改寄存器的值到 InitContext 的 DLLSaveXXX 域；
- 将当前初始化上下文的备份的指针保存到 InitContext.OuterContext 域；
- 初始化 InitContext.InitCount；
- 初始化 InitContext.DLLInitState=Reason+1(标识.*EXE* 模块时占用了 0 值)；
- 异常管理，线程局部存储管理；
- 调用 DllProc，使用户代码得到处理入口的机会；
- 如果不是 DLL_PROCESS_ATTACH 入口，则跳转到_Halt0()做结束化理；
- 单元初始化。

5.3.4　.DLL 的结束化过程

内部例程_Halt0()同时扮演了.EXE 与.DLL 结束化过程的角色。

每一次操作系统向.DLL 的入口传入参数并调用时，最终都会执行到_Halt0()。也就是说，.DLL 中会多次调用_Halt0()，其准确的数量为：

```
2 + DLL载入后创建的线程数*2 + DLL载入前创建的线程数*1
```

_Halt0()总是在_StartLib()调用结束时被执行到，因此，_Halt0()的具体行为取决于_StartLib()中对系统参数的设置和调整。而_StartLib()例程的内部逻辑相对复杂，以不同的 Reason 入口参数调用_StartLib，主要功能的执行情况如表 5-3。

表 5-3　_Halt0()功能的执行情况

功　能	PROCESS_ATTACH	THREAD_ATTACH	THREAD_DETACH	PROCESS_DETACH	备注
InitContext.DLLSaveXXX置值	Y	Y	Y	Y	
备份信息到OuterContext	Y	Y	Y	Y	
InitContext.InitCount置值	0	0	0	单元初始化表长	
初始化异常管理机制	Y	Y	Y	Y	
DLLInitState置值	2	3	4	1	

续表

功　能	PROCESS_ATTACH	THREAD_ATTACH	THREAD_DETACH	PROCESS_DETACH	备注
调用线程局部存储管理例程	InitProcessTLS	InitThreadTLS	N	ExitProcessTLS	①
调用DllProc()	N	Y	Y	Y	
调用_Halt0()	N	Y	Y	Y	
调用InitUnits()	Y	Y	Y	Y	②

例程_Halt0()中与DLL相关的代码有：

```
procedure _Halt0;
var
  P: procedure;
begin
  // ...
  if ErrorAddr <> nil then
  begin
    MakeErrorMessage;
    WriteErrorMessage;
    ErrorAddr := nil;  // 这行代码是为.DLL准备的，对.EXE没有意义
  end;

  while True do
  begin
    if (InitContext.DLLInitState = 2) and (ExitCode = 0) then
      InitContext.InitCount := 0;

    FinalizeUnits;

    // 下面这一段代码，可以参考.EXE结束化处理中的相关描述
    if (InitContext.DLLInitState <= 1) or (ExitCode <> 0) then
    begin
      if InitContext.Module <> nil then
        with InitContext do
        begin
          UnregisterModule(Module);
          if (Module.ResInstance <> Module.Instance) and
            (Module.ResInstance <> 0) then
```

① 这些例程指针保存在全局数组常量TlsProc中，在StartLib()的入口参数通过寄存器ECX传入(这时它通常被叫做InitTLS数组)。这其中，ExitThreadTLS()例程将延迟到DLLProc()调用结束后才被调用。

② InitUnits()执行的具体操作与InitContext.InitCount值是有关的，因此尽管每一种入口最终都将调用InitUnits()，但DLL_PROCESS_DETACH参数入口与其他入口并不一样。

```
          FreeLibrary(Module.ResInstance);
        end;
    end;

    // ...

    // 如果是DLL, DllInitState = 1..4
    if InitContext.DllInitState <> 0 then
      ExitDll;

    // ...
  end;
end;
```

如果是以 DLL_PROCESS_ATTACH 为入口，则必然是通过在库项目文件中的"end."行进入_Halt0()的。此时，系统已经调用过 InitUnits()，且 InitCount 存放着初始化过的单元数。如果初始化过程正常(没有修改 ExitCode)，则需要将 InitCount 置 0，以避免调用 FinalizeUnits 时做不必要的单元结束化处理。事实上，在_Halt0()中的这段代码：

```
    if (InitContext.DLLInitState = 2) and (ExitCode = 0) then
      InitContext.InitCount := 0;

    FinalizeUnits;
```

与下面的代码是等义的：

```
case InitContext.DLLInitState of
  0 :             // .EXE, 单元结束化
    FinalizeUnits;
  3, 4 :          // .DLL, 且Reason = DLL_THREAD_ATTACH 或 DLL_THREAD_DETACH
    { 由于在_StartLib()中设置InitCount=0, 所以是否调用FinalizeUnits()无意义 };
  2 :             // .DLL, Reason = DLL_PROCESS_ATTACH
    if ExitCode <> 0) then
      FinalizeUnits
    else
      { 进程载入时, 不应该执行单元结束化 };
  1 :             // .DLL, Reason = DLL_PROCESS_DETACH
    FinalizeUnits;
end;
```

对于任何参数的.DLL 入口调用，_Halt0()都会调用内部例程 ExitDll()来退出。过程 ExitDll()主要用于恢复 DLL 入口时的堆栈和寄存器现场。此外，由于 DllMain()实际上是一个返回 BOOL 值的函数，操作系统需要识别它的返回值并作进一步处理。因此，ExitDll()也根据全局变量 ExitCode 的值置 EAX。

5.3.5 DllProc 与 DllMain()的不同

Delphi 中没有 DllMain()。部分资料声称，Delphi 中的过程变量 DllProc 用以替代 DllMain()。

但这两者之间存在很多不同。

首先，DllMain()被用来接管 DLL 的入口，所以在 DllMain()之前应该是没有代码的。然而在 DllProc 被执行之前，已经有_InitLib()和_StartLib()被执行过了。因此，DllProc()面对的将不再是最初 DLL 的加载现场——栈以及一些全局变量已经被处理过了。

其次 DllProc()的入口中不再有 hInstance 参数，如果要使用它，可以访问全局变量 HInstance。

最后，也是最重要的一点：基于 Delphi 的机制，在 DllProc 中不可能得到处理 DLL_PROCESS_ATTACH 的机会。因为 Delphi 的用户代码是从单元初始化开始的，而单元初始化例程却是在 DLL_PROCESS_ATTACH 处理完之后，才会被_StartLib()调用的。这意味着没有任何用户代码，可以赶在内核试图以 DLL_PROCESS_ATTACH 为参数调用 DllProc 之前，将 DllProc 初始化为有效值。

作为弥补，Delphi 提供了一种技巧让开发者得到处理 DLL_PROCESS_ATTACH 的机会。例如下面这个项目：

```
library Project1;
uses SysUtils;
begin              //call @InitLib
  sleep(10);
end.               //call @Halt0
```

begin 行相当于 DllMain()入口，在这里，编译器加入了调用_InitLib()的代码。因此，当_InitLib()以及_StartLib()被调用完成后，程序流程会回到“sleep(10);”处，最后再通过编译在“end.”行中的“call @Halt0”退出。

由于在_StartLib()中有这样的代码：

```
        //...
@@haveExe:
        MOV     EAX,[ESP+12]    // 取Reason值
        DEC     EAX             // 减1
        JNE     _Halt0          // 如果Reason - 1 <> 0，则跳转到_Halt0()，结束化
```

可见，只有以 Reason 参数为 DLL_PROCESS_ATTACH 时，流程才会回到项目文件中的

"end."行，其他情况下，都会直接跳转到_Halt0()。因此，这样编写 DllProc()是完全可行的[①]：

```
library SameDLLmain;

uses
  Windows, SysUtils, Dialogs;

procedure Same_DllMain(Reason: Integer);
begin
  case Reason of
    DLL_PROCESS_ATTACH,
    DLL_THREAD_ATTACH,
    DLL_THREAD_DETACH :
      ShowMessage(Format('HInst:%.8X , Reason: %.8x', [HInstance, Reason]));
    DLL_PROCESS_DETACH :
      ( 小心调用API或其他例程，可能导致不可测的后果 );
  end;
end;

begin
  // 下面代码只会在DLL装载时被执行到一次，此时系统的状况为：
  //   1. HInstance为当前模块的实例句柄
  //   2. 内部模块表已经准备好
  //   3. 线程局部存储已经初始化
  //   4. 单元已经完成初始化
  if not assigned(DllProc) then  // 单元初始化中，可能已经给DllProc赋过值
    DllProc := @Same_DllMain;
  Same_DllMain(DLL_PROCESS_ATTACH);
end.
```

5.3.6　动态链接库的内核最小化

如果试图以类似于 C 语言或者汇编语言的方式来写一个 DLL，可以将 Delphi 中与动态链接库相关的代码大量简化，甚至可以用手工的方式来接管 Delphi 的入口代码。下例可以实现一个极小化的 DLL：

```
library MinimumDll;

function MessageBox(hWnd: LongWord; lpText, lpCaption: PChar;
```

① 事实上，自 Delphi2.0 开始，DllProc 就存在这样的问题。下面的方法是官方推荐的技巧。

```
  uType: LongWord): Integer; stdcall; external user32 name 'MessageBoxA';

const
  MB_ICONINFORMATION = $00000040;

var
  sArr : array [0..3] of pChar
       = ('PROCESS_DETACH', 'PROCESS_ATTACH', 'THREAD_ATTACH',
          'THREAD_DETACH');

procedure Same_DllMain(Reason: Integer);
begin
  MessageBox(0, sArr[Reason], 'MinimumDll', MB_ICONINFORMATION);
end;

procedure ExitDll(EAX : LongBool);
asm
  LEAVE
  RET 12
end;

begin
  // 在这里，可以开始写类似于DllMain()的处理代码，但入口的情况如下
       { ->     EAX Inittable      }
       {        [EBP+8] Hinst      }
       {        [EBP+12] Reason    }
       {        [EBP+16] Resvd     }

  // 这里示范如何实现与Delphi的DllProc类似的调用
  asm
    mov eax, [EBP+12]
    mov ecx, [EBP+16]
    call Same_DllMain
  end;

  // 这里是类似于DllMain()的返回方式
  ExitDll(True);
end.
```

编译上面的项目，仅需要在最简化内核中加入如下代码：

```
interface
// ...
type
  TInitContext = Integer;  // 未用
```

```
procedure _InitLib;

implementation

// ...
procedure _InitLib;
asm
end;
```

需要注意的是，改变 TInitContext 的类型定义，会影响入口代码中在栈上分配空间的大小，但是不会影响程序的正常运行。因为在最小化内核中，并不需要使用 InitContext。

在随书光盘中，提供一个为编写 DLL 而改写的最小化内核及其示例代码 *MiniDll.dpr*。使用这个内核，可以用 Delphi 标准的方式开发动态链接库，而不必像上面的例子那样去重写与 DllMain() 相关的代码。

5.4　Delphi 的动态链接库：包

Delphi 的包是一个扩展名为 .BPL 的动态链接库(DLL)文件。不过，Delphi 在编译器一级为 *.BPL* 加入大量的代码，并提出一整套相关的约定，使得它成为了一个极其复杂的系统。

Delphi 实现包的主要目的是要解决这样一些问题：

- 在 IDE 中添加和管理组件，包括第三方的组件；
- 在 IDE 中实现插件及其相关的功能；
- 在应用程序中发布，使用类似于 DLL 的代码复用技术来减少模块的大小；
- 在多个模块之间，共享类、组件和变量。

这里的前两个问题主要以设计期包来解决，后两个则是运行期包的责任。

5.4.1　包的主要规则

■ 包文件的生成规则

一个包的工程文件被成功编译后，生成的文件包括表 5-4 中的几种类型。

如果一个包使用{$IMPLICITBUILD ON}条件编译，那么每次使用到这个包，编译器总是尝试重新编译它。因此商业发布的以及限制改写的包文件通常应使用{$IMPLICITBUILD OFF}编译。

表 5-4 包文件成功编译后所生成的文件类型

扩展名	描　述
.dcu	必要时，每个被包含的单元都会生成自己的.dcu文件
.dcp	二进制编译期的影像(MAP)文件，由包头信息和数据体(串接所有被包含单元的.DCU文件)构成
.bpl	编译后的二进制文件

■ 包内单元的重复性约定

- 一个单元 uses 节中列出的单元，会被(间接)包含到包文件中；
- 一个应用不能载入两个包含(直接或间接)有相同单元文件的包。

如果单元是弱连接的，则重复包含它的多个包可以被同一个应用载入。

单元的重复性检测是在应用程序运行期发生的事。编译器只决定在编译期将哪些单元放在包中，而并不考虑多个包中是否有相同的单元。

要使单元不被包含，可以在单元中加入编译条件{$DENYPACKAGEUNIT OFF}，这样，即使包中直接或间接使用了这个单元，该单元也不会被编译到包中，自然也不会参与重复性检测。

尽管包的项目文件名在 Delphi 内部也被描述成单元名，但并不参与重复性检测。

■ 弱连接的相关约定

如果一个单元使用{$WEAKPACKAGEUNIT ON}条件编译，则表明它使用弱连接方式连入包文件中。这种情况下，包的.dcp 文件仍然会串接该单元，但只有在包中的其他单元引用了弱连接单元时，该单元才会被包含到.bpl 中。

因此，弱连接单元实际上是无法实现代码复用的。如果应用程序引用了弱连接单元，则一份该单元的代码拷贝将会被编译到.EXE 中；如果该应用程序的一个包也使用了这个单元，那么在包中也包含了这个单元的一份代码。并且，应用程序模块与包模块使用各自的代码拷贝来运行。

基于上述的理由，弱连接单元不能有初始化节和结束化节，因为在应用程序模块和包模块

中分别初始化这个单元的行为是没有道理的。也不能有在应用程序这一级别上共享的全局变量，因为不同的模块使用着各自的数据和代码拷贝。

■ 包的类型约定

设计期包：使用{$DESIGNONLY ON}条件可以编译仅适用于 Delphi IDE 环境的包。通常使用设计期包来制作组件属性编辑器，以及 IDE 上的专家或插件。

运行期包：使用{$RUNONLY ON}条件可以编译适用于普通应用程序在运行期动态载入的包。

不使用其他包中的共享数据的包：使用{$IMPORTEDDATA OFF}条件，可以编译这种类型的运行期包或设计期包。这使得包文件在内存访问上更有效率。

不需要重新编译的包：使用{$IMPLICITBUILD OFF}条件，可以使编译的运行期包或设计期包不被重新编译。这样可以不带源程序地分发包文件给其他开发人员。

5.4.2　Delphi 中的包与普通 DLL 的区别

尽管操作系统认为包与普通的 DLL 是同类型的文件，但是源于 Delphi 中不同的实现机制，二者是有区别的。

■ 装载包和.DLL 的例程不同

应当使用 API：LoadLibrary()来装载 DLL，使用例程 LoadPackage()来装载包。如果包不需要单元初始化凡不必检测重复单元(例如仅用于载入资源的包)，那么使用 LoadLibrary()来载入包也是安全的。无论使用哪种方法载入包，包都会被添加到操作系统的模块表中。

■ 包与 DLL 的初始化、结束化存在较大的差异

DLL 的初始化与结束化都在 DllMain()中完成。而包不是这样。包的 DllMain()函数仅用于安排线程局部存储和操作内部模块表，这仅仅是模块初始化、结束化工作的一部分。

包的另外一部分初始化、结束化代码分布在如下的例程中(表 5-5)。

表 5-5 包的一部分初始化、结束化程序代码

	SysUtils.pas	SysInit.pas	System.pas	备 注
初始化	LoadPackage()	_PackageLoad()	_PackageLoad()	需要从Host程序中，调用SysUtils.pas中的例程来进行初始化和结束化。
结束化	UnloadPackage()	_PackageUnload()	_PackageUnload()	

包的初始化包含了一个检测复重单元的操作，由 *Host* 程序调用 *SysUtils.pas* 中的 LoadPackage()来实现。

系统全局变量 DllProc 对于包是无意义的。

DLL 结束化时会调用 Halt0()，但包结束化时不会这样做。

■ 对内部模块表的使用

除了将自身安装在内部模块表中，DLL 基本不对内部模块表做其他的操作。

包将使用内部模块表来进行单元重复性检测。即使是使用 LoadLibrary()将包加载到进程，该包仍然会被注册到内部模块表中。这也就是说，使用 LoadLibrary()加载时虽然能避免自身进行单元重复性检测，却不能避免其他模块对该模块(所含的单元)进行重复性检测。

5.4.3 包的 DllMain()

包的入口代码只有一行，无条件跳转到_InitPkg()例程。跳转指令维护了堆栈结构的完整性，因此，在_InitPkg()入口处，具有跟 DLL 入口点完全一致的堆栈现场。也就是说，_InitPkg()可以直接看作 DllMain()例程。

```
// code from SysInit.pas
function _InitPkg(Hinst: Longint; Reason: Integer;
  Reserved: Pointer): Longbool; stdcall;
begin
// ...
  if @TlsLast <> nil then
    TlsProc(Reason);
  if Reason = DLL_PROCESS_ATTACH then
    InitializeModule
  else if Reason = DLL_PROCESS_DETACH then
    UninitializeModule;
  _InitPkg := True;
end;
```

初始化例程_InitPkg 的源码很简单，只根据 Reason 处理线程局部存储和内部模块表而已。

5.4.4　包的载入例程 LoadPackage()

包的一部分秘密藏在 *SysUtils.pas* 中——仅仅_InitPkg()是不可能实现那样复杂的包系统的。

尽管.*BPL* 是标准的 DLL，但是并不应该用 LoadLibrary()来加载使用，而应该使用 *SysUtils.pas* 中的载入例程 LoadPackage()。LoadPackage()除了用 SafeLoadLibrary()将.*BPL* 装入内存[①]，还调用 InitializePackage()例程来进一步初始化。

```
// code from SysUtils.pas
function LoadPackage(const Name: string): HMODULE;
begin
  Result := SafeLoadLibrary(Name);
  if Result = 0 then
  begin
    raise EPackageError.CreateResFmt(@sErrorLoadingPackage,
      [Name, SysErrorMessage(GetLastError)]);
  end;
  try
    InitializePackage(Result);
  except
    FreeLibrary(Result);
    raise;
  end;
end;
```

5.4.5　真正的初始化例程 InitializePackage()

InitializePackage()用于完成两项工作：

- 通过编译在包的 PE 头结构中的资源对包内单元进行重复性检测；
- 通过查找和调用输出例程来初始化。

① SafeLoadLibrary()调用 LoadLibrary()。

■ 包内单元的重复性检测

InitializePackage()调用 CheckForDuplicateUnits()检测重复单元。这个例程首先从包的资源中取出资源“PACKAGEINFO”，这是一个 TPackageInfoHeader 类型的数据块，然后按照包的规则进行检测。

CheckForDuplicateUnits()的检测过程是：先对指定包列出所有请求(Requires)的包，并递归调用 CheckForDuplicateUnits()。接下来，列举出指定包所包含(Contains)的单元，并调用 IsUnitPresent()例程检查是否已存在单元，如果是，则引发一个 EPackageError 异常。如果单元是弱连接的，或者是项目文件(*.dpk*)，则忽略对它的 IsUnitPresent 检查。

```
// code from SysUtils.pas
function IsUnitPresent(RC: Byte; UnitName: PChar; Module: HMODULE;
  const ModuleName: string; var UnitPackage: string): Boolean;
var
  // ...
begin
  Result := True;
  if (StrIComp(UnitName, 'SysInit') <> 0) and
    (StrIComp(UnitName, PChar(ModuleName)) <> 0) then
  begin
    // ...
```

这里的两个检测用于忽略对 *SysInit.pas* 和包的项目文件(*.DPK*)的检测。看起来，这里存在一个安全隐患：由于入口中的 ModuleName 是通过 API：GetModuleFileName 来获取的，这个 API 取得的是磁盘上的文件名，而用来比较的 UnitName 却是编译期放在文件资源表中的，因此如果载入的包文件被更名，则 IsUnitPresent()对项目文件(*.DPK*)的检测将会存在差错。

但是事实上，由于在调用 IsUnitPresent()之前，已经通过 Flags 判定并过滤掉了 *.dpk* 单元，所以上述的问题实际上并不会发生。但是，也看不出有再次检测包文件名的必要了。

接下来的代码中，判定单元文件重复的条件是：

```
// code from SysUtils.pas, IsUnitPresent()
// ...

// UName^.Name是从内部模块表中列举到的被测单元
// UnitName是例程入口参数中的待测单元
with UName^ do
```

```
if ((HashCode = HC) or (HashCode = 0) or (HC = 0)) and
  ((Flags and $06) = 0) and (StrIComp(UnitName, Name) = 0) then
begin
  // ...
```

这里使用 Hash 算法为单元文件名进行编码，用于快速地比较文件名是否相同。被测单元或待测单元之一的 HashCode 为 0 时，表明强制进行单元名检测。条件“(Flags and $06)”则用于检测被测单元是否为弱连接的或者项目文件，如果是，则忽略检测。

项目文件名不参与重复性检测将导致这样一种情况发生：在同一应用载入的两个包中，A 包可以有与 B 包项目文件名相同的单元文件，反之亦可。例如下面的两个包，可以被一个应用同时载入：

```
//包一
package Package_A;

requires
  rtl;

contains
  Package_B in ' Package_B.pas';
end.

//包二
package Package_B;

requires
  rtl;

contains
  Package_A in ' Package_A.pas';
end.
```

如果在 IsUnitPresent()中检测到重复单元，该例程返回 True，并将存在重复单元的包名填入变量参数 UnitPackage。

■ 调用输出例程初始化

InitializePackage()例程调用 API：GetProcAddress()来查找并调用包中的输出例程“Initialize”，从而完成初始化。

```
// code from SysUtils.pas, InitializePackage()
// ...
```

```
  @PackageLoad := GetProcAddress(Module, 'Initialize'); //Do not localize
  if Assigned(PackageLoad) then
    PackageLoad
// ...
```

5.4.6 包的卸载例程 UnloadPackage()

同样位于 *SysUtils.pas* 中的 UnloadPackage() 例程，只是简单地调用 FinalizePackage()进行结束化处理，并调用 API：FreeLibrary()将包从进程空间中移除。

FinalizePackage()例程查找并调用包的输出例程“Finalize”以完成包的结束化。

5.4.7 包的基本输出例程

对于任何一个包文件，编译器都会使它输出多个例程，例如在包的装载和卸载过程中使用到的“Initialize”和“Finalize”。对于一个空的包文件：

```
package NullPackage;

requires
  rtl;

end.
```

使用工具 TDump 查看它编译后的.*bpl*：

```
C:\>tdump -m -ea:v NullPackage.bpl
Exports from NullPackage.bpl
  7 exported name(s), 7 export address(s).  Ordinal base is 1.
  Sorted by RVA:
    RVA      Ord. Hint Name
    -------- ---- ---- ----
    0000124C    1 0006 @GetPackageInfoTable
    0000124C    2 0003 @Nullpackage@@GetPackageInfoTable$qqrv
    00001254    3 0002 @Nullpackage@@PackageLoad$qqrv
    00001254    7 0005 Initialize
    00001260    4 0001 @Nullpackage@@PackageUnload$qqrv
    00001260    6 0004 Finalize
    0000126C    5 0000 @Nullpackage@initialization$qqrv
```

通过对比可发现，Initialize 与 PackageLoad()使用同一地址[①]。类似地，Finalize 和 GetPackageInfoTable 也使用与 PackageUnload()、GetPackageInfoTable()相同的地址。

表 5-6 对照地列出包的基本输出例程与 *System.pas* 中源码的关系。

表 5-6　包的基本输出例程与 *System.pas* 中源码的关系

RVA	输出名	内部输出名[②]	*System.pas*
00001260	Finalize	@Nullpackage@@PackageUnload$qqrv	_PackageUnload()
00001254	Initialize	@Nullpackage@@PackageLoad$qqrv	_PackageLoad()
0000124C	@GetPackage InfoTable	@Nullpackage@@GetPackageInfoTable$qqrv	编译器生成[③]
0000126C		@Nullpackage@initialization$qqrv	入口代码地址[④]

5.4.8　内部例程_PackageLoad()与_PackageUnload()

这两个内部例程在 *SysInit.pas* 单元中声明如下：

```
// define in SysInit.pas
procedure _PackageLoad(const Table: PackageInfo);
procedure _PackageUnload(const Table: PackageInfo);
```

但在 *SysUtils.pas* 中，却是使用下面这种方式来调用它们的导出例程的：

```
// code from SysUtils.pas, InitializePackage()
// ...

  PackageLoad := GetProcAddress(Module, 'Initialize'); //Do not localise
  if Assigned(PackageLoad) then
    PackageLoad
// ...
```

入口参数中的 Table 是编译器生成的初始化表地址，是用户代码不可见的。因此为了传入

① 这里使用的是相对虚地址(RVA)，有关相对虚地址的内容，参见第 6 章“Delphi 的积木艺术(PE)”。

② 内部输出名的命名格式含义参见第 6 章“.EXE 和.BPL 中的导出表”小节。

③ 在 *System.pas* 中 *GetPackageInfoTable* 只是一个类型定义，编译器会生成一个输出项，使它在 *EAX* 中返回 *PackageInfoTable*。

④ 在.dpk 文件中，并没有初始化节。但编译器生成名为“@Nullpackage@initialization$qqrv 件的入口代码。API：LoadLibrary()并不使用这个输出项来查找入口点。

这个参数，编译器并没有直接导出_PackageLoad()的地址，而是导出了一段前导代码的地址。例如初始化时_PackageLoad()的导出：

```
@PackageLoad:
04A11254 B92C12A104      mov eax,$04a1122c
04A11259 E87EFFFFFF      call -$00000082     // call @PackageLoad
04A1125E C3              ret
```

地址$04A11254 的 RVA 是$00001254，即'Initialize'的导出地址。这个地址上的代码，仅用于向 EAX 送入 Table 参数。接下来"call -$00000082"，便是调用 *SysInit.pas* 中_PackageLoad()。代码如下：

```
// define in SysInit.pas
procedure _PackageLoad(const Table: PackageInfo);
begin
  System.PackageLoad(Table, @Module);
end;
```

这样，*SysInit.pas* 将单元局部变量 Module 的地址传给 *System.pas* 中的_PackageLoad()，从而完成初始化。

Sytem.pas 中的例程_PackageLoad()与_PackageUnload()主要负责单元初始化与结束化：

```
procedure _PackageLoad(const Table : PackageInfo; Module: PLibModule);
  // ...
begin
  // 备份旧的初始化上下文，并为当前操作初始化一个上下文
  // ...
  try
    InitUnits;
  // ...
end;

procedure _PackageUnload(const Table : PackageInfo; Module: PLibModule);
// ...
```

包不会调用_Halt0()执行结束化。但是，HOST 进程调用_Halt0()结束化时，会通过访问内部模块表对包进行结束化操作。

5.4.9 包的内核最小化

在随书光盘中提供了一个包的最小化内核。对于使用这个内核编译的包来说，由于内核并

没有实现_PackageLoad()和_PackageUnload 的代码，因此包中的单元初始化和结束化没有意义；而且由于内核中的_InitPkg() 不操作内部模块表，因此加载该包的模块也不能对它进行单元重复性检测。

此外，这个内核中也没有包含基于线程的上下文备份，因此用它编译的包中的代码和数据是非线程安全的(例如线程局部存储无效)。

5.5　其他

本小节中的内容，部分是对前面所述内容的补充描述。

另一部分，则是我在分析内核时对一些问题的不十分成熟的思考。对于这一部分内容，我希望读者能有甄别地阅读：一些情况下，我无法对相关的问题得出明确的答案，甚至有些思考也是存疑的。

你需要在已经很好地理解了本章中的内容以后，才适宜继续阅读本节。当然，你也可以跳过本节，这不会影响到对后续篇章的理解。

5.5.1　初始化上下文中 OuterContext 域的使用

对于使用“带运行期包”方式编译的模块来说，同一进程中不同的模块都使用同一个 *rtl.bpl*。因此，不同的模块都在访问同一个系统局部变量 InitContext。然而，按照 *System.pas* 中的代码来分析，不同的模块应该拥有各自独立的 InitContext。因此，*System.pas* 需要用 OuterContext 域来保存“上一个”模块使用的 InitContext，并在当前模块使用完之后即时恢复它。

对于不使用“运行期包”的模块，.DLL 的不同线程进入 DllMain()函数时，使用的寄存器信息不同，因此也需要使用 OuterContext 来暂存当前线程的信息。

此外，InitContext 中除了保存初始化和结束化信息之外，还保存着当前模块的运行期信息，例如引发异常时的代码现场信息。因此，因其他任何原因使用 InitContext 时，都应当即时保存和恢复该变量。

因为 InitContext.OuterContext 域是一个指针，需要在该域中备份信息，则必须由调用者代码自行初始化一个 TInitContext 类型的记录。因此：

- .DLL 通过入口代码，在调用者进程的堆栈中开辟的空间；
- 包通过_PackageLoad()例程中的局部变量开辟的空间；

被用来暂存该记录。

包在DllMain()函数中并不使用InitContext。因此，它不需要为每一个线程保存寄存器信息，而仅是在_PackageLoad()和_PackageUnload()例程中操作属于当前模块的唯一一份InitContext。

5.5.2 入口代码中的堆栈使用深入分析(内存现场)

■ 入口代码之前的初始栈

对于EXE模块，它对应于一个主线程；而对于DLL/BPL模块来说，它可能对应于主线程或者进程中新创建的任何线程。因此，EXE模块的入口代码所面对的是进程的主线程的初始栈，而DLL与BPL中的入口代码则将面对未知多个新创建线程的初始栈。

初始栈中有什么呢？

对于EXE来说，入口代码被作为一个无参数main()函数调用，因此进入入口代码时栈上是没有传入参数的。对于DLL和BPL来说，入口代码被作为dllmain()函数，以stdcall约定调用，因此这时栈上会有三个参数，分别是：hModule,dwReason和lpvReserved。

参数之前并不会立即到达栈底。因为对于操作系统来说，入口代码并不是线程执行的第一行代码，而那些在入口代码之前的、操作系统的代码也将使用线程栈。但在用户代码看来，这些太古时代的数据看不到，也不需要用到。

■ Delphi入口代码中堆栈初始分配的内容

在EXE和DLL的初始化代码中都有直接分配栈的操作。对于EXE来说，是分配了一个Delphi的异常帧结构TExcFrame；对于DLL来说，除了TExcFrame之外，还分配了一个TInitContext类型的记录。

在没有任何官方的资料来说明初始化代码中分配的这些栈空间的数据类型及具体作用的情况下，上述结论是如何得到的呢？

是例程_StartLib()中的如下代码给我们提供了相关的信息(请注意加粗的代码行)：

```
procedure _StartLib;
asm
// ...
        { Save the current init context into the stackframe of our caller }
```

```
        MOV     ESI,offset InitContext
        LEA     EDI,[EBP- (type TExcFrame) - (type TInitContext)]
        MOV     ECX,(type TInitContext)/4
        REP     MOVSD
// ...
```

在调用_InitLib()例程调用_StartLib()之前并没有修改过 EBP 寄存器，也就是说 EBP 寄存器的值来自于模块初始入口代码：

```
Project2.dpr.19: begin
00B81FD4 55              push ebp
00B81FD5 8BEC            mov ebp,esp
00B81FD7 83C4C4          add esp,-$3c
00B81FDA B85C1FB800      mov eax,$00b81f5c   // 传入单元初始化表地址
00B81FDF E8F435FFFF      call @InitLib       // 调模块初始化例程
```

因此可以得到结论，在代码：

```
        LEA     EDI,[EBP- (type TExcFrame) - (type TInitContext)]
```

中所操作的，正是入口代码：

```
00B81FD7 83C4C4          add esp,-$3c
```

中所分配的栈空间的首地址，亦即是“入口代码在栈上初始分配空间的头部”。而 SizeOf(TExcFrame)+SizeOf(TInitContext)大小，就是这些初始分配空间的长度值：$3C 字节。如果尝试在 DLL 模块的最小化内核中修改 TInitContext 类型的定义，将会使得 $3c 这个长度值发生变化。这也正好旁证了上述的推论。

(不同版本的)编译器内部约定了 TExcFramer 的类型声明，因此在代码中修改 TExcFramer 的类型声明是没有意义的。

■ 栈中的数据结构是被如何使用的

EXE 在模块初始栈用 TexcFramer 记录构造一个异常帧，以使得即使程序崩溃，系统仍然可以进入到一段可由用户代码控制的异常处理程序中。这个异常处理程序是 _ExceptionHandler()，它通过 SetExceptionHandler()例程在_StartExe()中被安装：

```
procedure _StartExe(InitTable: PackageInfo; Module: PLibModule);
begin
  // ...
  SetExceptionHandler;
  IsLibrary := False;
  InitUnits;
end;
```

```
procedure SetExceptionHandler;
asm
  XOR    EDX,EDX                 { using [EDX] saves some space over [0]     }
  LEA    EAX,[EBP-12]
  MOV    ECX,FS:[EDX]            { ECX := head of chain                      }
  MOV    FS:[EDX],EAX            { head of chain := @exRegRec                }

  MOV    [EAX].TExcFrame.next,ECX
  MOV    [EAX].TExcFrame.desc,offset _ExceptionHandler
  MOV    [EAX].TExcFrame.hEBP,EBP
  MOV    InitContext.ExcFrame,EAX
end;
```

如前所述，在_StartLib()中，[EBP-(type TExcFrame)-(type TInitContext)]指向DLL模块的“入口代码在栈上初始分配空间的头部”。同样的道理，在_StartExe()中，[EBP-(type TExcFrame)]就指向EXE模块的这个位置。TExcFrame记录的长度为$10字节，然而在SetExceptionHandler()中却使用[EBP-12]访问这个异常帧。这显得很奇特：有四个字节是未用的。

但事实上的确如此：在EXE模块中，保留的16字节中只有12字节被用。我相信这是代码上的小BUG。不过对于操作系统来讲，在异常链表中的每一个帧只需要12个字节，多出来的部分是Delphi的扩展，因此操作系统的SEH机制并不会认为这是个错误。关于这些内容的细节，请参见第8章的“顶层异常处理”小节。

然而DLL在模块初始化中分配的$3C字节的栈空间使用起来却没有这样简单。

首先，模块中声明了内核局部变量InitContext。然而，事实上应该为每一个线程创建一个该变量的备份。这个备份应该存在于(HOST进程的)该线程所管理的内存空间里，这样才可以消除其它线程结束化所带来的影响。保存这个InitContext变量的另一层意义在于：只有这样，在DllMain()执行结束后退出DLL的同时，才能为操作系统调用DllMain()的内核代码恢复正确的堆栈和寄存器值。

因此在_StartLib()中有代码来保存当前的InitContext变量：

```
procedure _StartLib;
asm
// ...
      { Save the current init context into the stackframe of our caller }

      MOV      ESI,offset InitContext
      LEA      EDI,[EBP-(type TExcFrame)-(type TInitContext)]
      MOV      ECX,(type TInitContext)/4
```

```
        REP     MOVSD
// ...
```

而在 ExitDll()中也有代码来恢复它：

```
procedure       ExitDll;
asm
// ...
        { Restore the InitContext }
        MOV     EDI, offset InitContext
        MOV     EBX,[EDI].TInitContext.DLLSaveEBX
        MOV     EBP,[EDI].TInitContext.DLLSaveEBP
        PUSH    [EDI].TInitContext.DLLSaveESI
        PUSH    [EDI].TInitContext.DLLSaveEDI

        MOV     ESI,[EDI].TInitContext.OuterContext
        MOV     ECX,(type TInitContext)/4
        REP     MOVSD
// ...
```

在_StartLib()中复份完当前的线程上下文后，直到 ExitDll()之前，模块中的代码都只操作模块的变量 InitContext。而在这些操作的初起，是_StartLib()所做的变量初始化：

```
procedure       _StartLib;
asm
// ...
        { Save the current init context into the stackframe of our caller }
// ...

        POP     InitContext.DLLSaveEDI
        POP     InitContext.DLLSaveESI
        MOV     InitContext.DLLSaveEBP,EBP
        MOV     InitContext.DLLSaveEBX,EBX
        MOV     InitContext.InitTable,EAX
        MOV     InitContext.Module,EDX
        LEA     CX,[EBP- (type TExcFrame) - (type TInitContext)]
        MOV     InitContext.OuterContext,ECX
// ...
```

除了 TInitContext 所用的$2C 字节之外，DLL 模块初始栈上剩下的*$10* 字节的 TExcFramer 记录，其使用方法与 *EXE* 模块是一致的。

5.5.3 再论入口代码

■ 操作系统与编译器的约定

在前面有许多次、许多节对模块入口代码的讨论，但是，都是以 Delphi 开发者为对象的，并且主要是针对内核源码来讲述的。

我主要是想避免讲操作系统。

但是，如果能从操作系统的角度来思考 Delphi 的模块入口代码，那么你可能看得更为清晰。

操作系统能从 PE 文件的头部取得模块代码的入口地址，这一点毋庸置疑。如果需要知道操作系统何时、以及如何取得入口地址，可以参见下一章“Delphi 的积木艺术(PE)”。这里，我们只假设操作系统取得了这个地址，并放在变量 lpEntryPoint_xxx 中。

接下来，操作系统按照约定将这个指针作为函数调用。用 Delphi 的方法来描述，代码如下：

```
type
  TLibEntry = function(hinstDLL:DWORD; fdwReason:DWORD;
    pvReserved:Pointer): BOOL; stdcall;
  TExeEntry = function : DWORD; stdcall;

var
  lpEntryPoint_DLL : TLibEntry;
  lpEntryPoint_EXE : TExeEntry;

begin
  // hModule := LoadModuleFromFile(aFileNameString);
  // lpEntryPoint_xxx := GetEntryPointFromModulePE(hModule);

  //for .DLL
  Result := lpEntryPoint_DLL(hModule, Reason, 0);

  //for .EXE
  Result := lpEntryPoint_Exe;
end;
```

也就是说，操作系统总是把.EXE 和.DLL 的入口指针当成一个函数来执行。当然，对于.EXE API：来说，通常是在 API：CreateProcess()中执行的，而对于.DLL 来说，装载时由 LoadLibrary()来执行函数。

操作系统会使得 lpEntryPoint_Exe 在新的进程中执行，而 lpEntryPoint_DLL 在调用者(HOST)进程中执行。两者有所不同，但不影响我们对流程的分析。

需要记住的是：如何调用入口指针是操作系统与编译器在模块执行流程方面的惟一约定。

一旦操作系统找到入口指针，就按照约定准备参数并调用，然后所发生的一切，是由编译器和用户代码来决定的。

接下来讨论模块中代码的执行流程。

■ 可执行程序的流程

Delphi 的可执行程序是从入口代码开始的，它实际上隐藏了一个响应 lpEntryPoint_Exe 指针的例程。将 Delphi 的入口以例程的形式恢复出来，可以是这样的：

```
procedure main;
var
  ExceptionFrame: TExcFrame;      // 在栈空间上分配310字节
begin

  //编译器在初始化代码中置EAX为初始化表地址，相当于如下汇编码
  _InitExe(PInitTable_FromEAX);
(*
  asm
    mov eax,$0044c8b0
    call @InitExe;
  end;
*)

  // 完成用户的代码，例如：
  // Application.Initialize;
  // Application.CreateForm(TForm1, Form1);
  // Application.Run;

  // 结束，可以使用_Halt0，或者直接调用TerminateProcess()
  ExitProcess(ExitCode);
end;
```

与这个过程出口处的“end;”如何编码是没有关系的。因为在 Delphi 中，进程使用专用的例程来中止或者退出，并不会通过“end;”再返回到调用者(操作系统)。

习惯使用 C 语言或者汇编语言的读者应该已经能把这个过程与 main()联系起来了。

■ 动态链接库的流程

与可执行文件一样，动态链接库也隐藏了响应 lpEntryPoint_DLL 的例程。把 DLL 文件的入口以 Delphi 例程的方式恢复出来，可以是这样的：

```
function DllMain(hinstDLL:DWORD; fdwReason:DWORD;
  pvReserved:Pointer): BOOL; stdcall;
var
  // 在栈空间上分配 510+92C=83C 字节
  ExceptionFrame: TExcFrame;
  TempContext: TInitContext;
begin
  _InitLib;

  // _InitLib()内部实现了类似于下面这样的一个逻辑
( if fdwReason = DLL_PROCESS_ATTACH then
    InitUnits
  else
  begin
    if Assigned(DLLProcEx) then
      DLLProcEx(fdwReason,0);

    if fdwReason = DLL_PROCESS_DETACH then
      FinalizeUnits;

    ExitDll;
  end;
)

  // 用户在项目文件中的初始化代码。例如:
  // DllProc := @Same_DllMain;
  // Same_DllMain(DLL_PROCESS_ATTACH);

  // 结束化处理并在_Halt0()中调用ExitDll;
  ExitDll;
end;
```

Delphi 的 DllMain()实际上总是用 ExitDll()退出的，在 ExitDll()中有负责修正堆栈的代码，使得这个 DllMain()退出时使用与 stdcall 约定一致的堆栈处理规则。

总体看来，Delphi 的这个 DllMain()与 C 语言或者汇编语言中的 DllMain 有着相同的效果。

■ 包文件的流程

在操作系统看来，包文件也是一个标准的动态链接库，也可以通过 API：LoadLibrary()来装载。Delphi 对包的特殊处理是一些内部的技巧，并不影响“操作系统把包文件视作动态链接库”这样一个事实，也不对操作系统调用包文件的流程构成任何影响。

包几乎没有入口代码，只有一行“jmp @InitPkg”使流程跳转到 *System.pas* 中的包文件初始化例程中。因此，对于包来说，_InitPkg()采用与操作系统的 DllMain()规范完全一致的例程接口来编写。

_InitPkg()是真正的 DllMain()函数。

第 6 章

Delphi 的积木艺术(PE)

Delphi 必须使自己编译出来的文件被操作系统识别。因此，编译器需要按照操作系统的约定，将模块编译成 PE 格式的文件。

按照约定，编译器可以将一部分工作交给操作系统的模块载入例程完成。载入例程所需要的信息也是通过 PE 格式的文件头来获取的。

Delphi 的编译器将一部分信息保存在某些节中，这使得用户代码能够得到并处理编译期才能知道的一些信息，也可以使不同的模块按同一约定交换数据。这种技巧在包(*.BPL*)的实现中被频繁使用。

Delphi 按照固定的 PE 文件头格式编译生成文件。本章介绍与 Delphi 相关的各种规则和数据结构，并不包括 PE 结构的全部。因此，有关 PE 结构的更多内容，可以参见本书附录中列举的资料。

6.1 PE 文件结构概要

从物理结构上来看，PE 文件非常简单。PE 文件可以分成四个部分，如图 6.1 所示。许多工具可以直观地显示出 PE 文件结构及其十六进制数据，例如 stud_PE 和 JIURL PEDump。

以 JIURL PEDump 为例，查看 Delphi7 的可执行文件 *Delphi32.exe* 如图 6.2 所示[1]。

(1) 此情况下，本章中将直接使用 Delphi7 安装目录中 Bin 子目录下的可执行文件作为分析实例。这时不再另行声明所使用的文件版本。

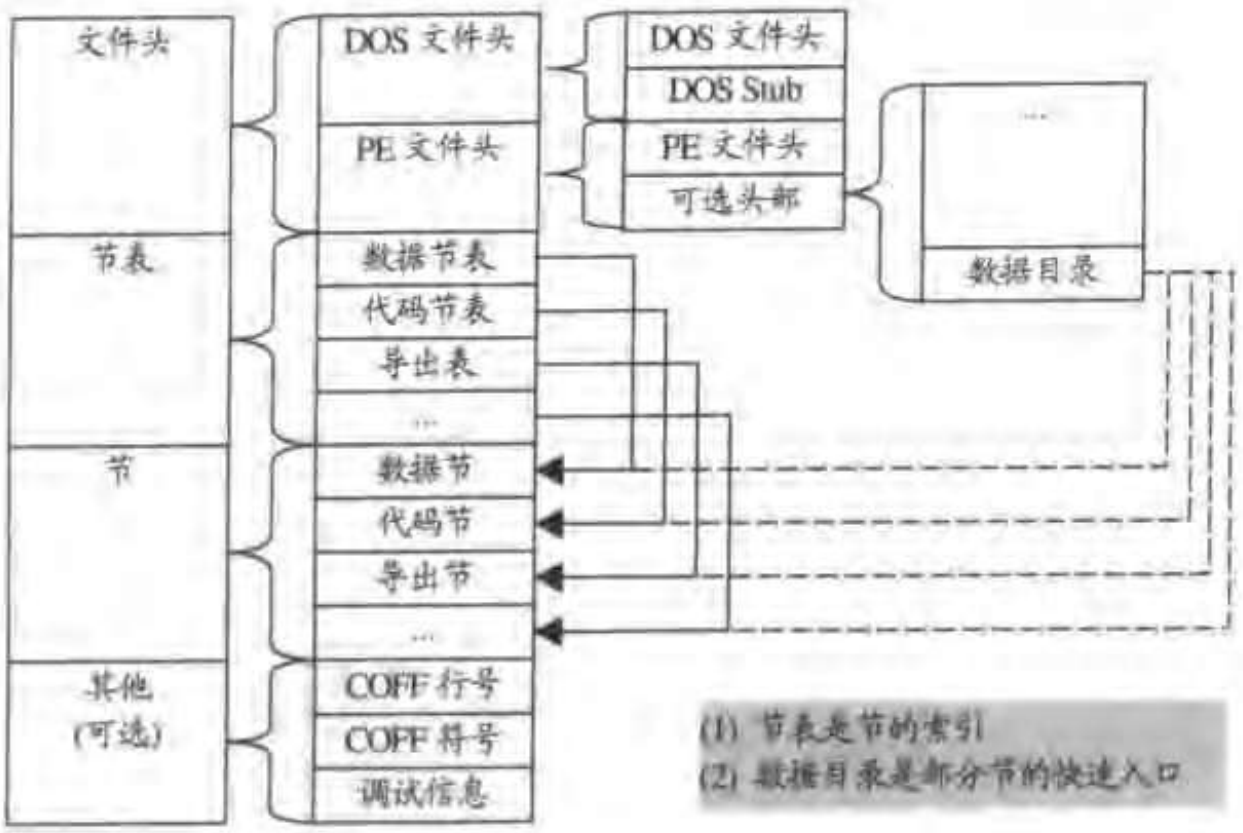


图 6.1　PE 文件结构的逆层分解

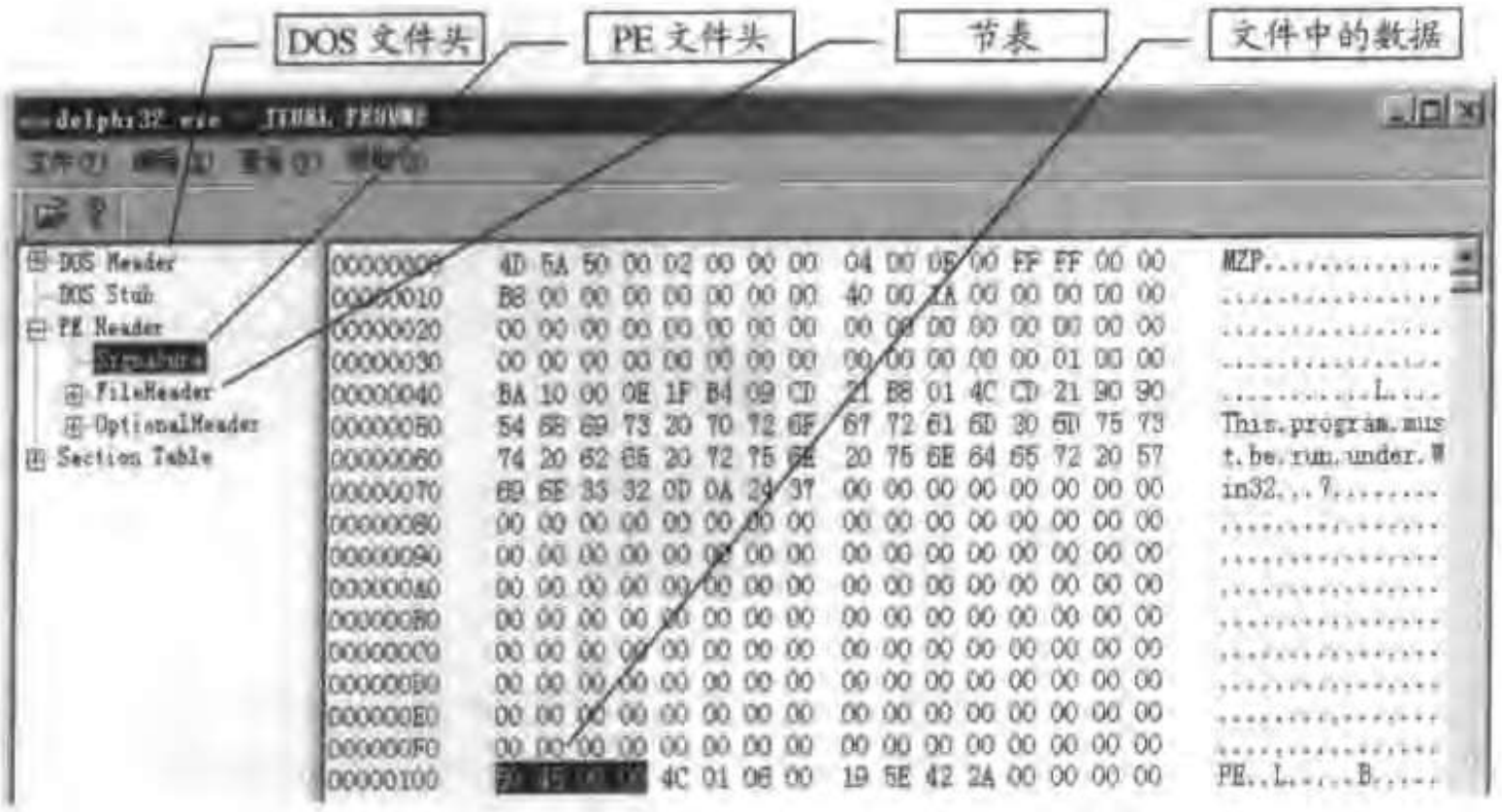


图 6.2　使用 JIURL PEDump 查看 PE 头的情况

6.1.1 文件头

在 Delphi 源代码的 *Windows.pas* 中，可以找到 DOS 文件头、PE 文件头和可选头部的定义[①]。

可以非常简单地取得 PE 头的位置和各个域的数据，如下列代码所示：

```
var
  p: PByte;
  Buff: Pointer;
  i, tempSize : Integer;
begin
  // ...
  // 将文件读到Buff所指向的内存块

  // DOS 头部
  p := Buff;
  writeln('DOS头部标志: $', IntToHex(PImageDosHeader(p)^.e_magic, 4),
    '  -->MZ');


  // PE 头部标志
  inc(p, PImageDosHeader(p)^._lfanew);
  writeln(' PE头部标志: $', IntToHex(PImageNtHeaders(p)^.Signature, 8),
    '  -->PE');


  // PE 头部
  with PImageNtHeaders(p)^.FileHeader do
  begin
    writeln(' PE头部的域 ==> ');
    writeln('      NumberOfSections  : ', NumberOfSections);
    writeln('         TimeDateStamp  : $', IntToHex(TimeDateStamp, 8));
  end;

  // PE 可选头部
  // ...

  // 修正可选头部的长度
  inc(p, PImageNtHeaders(p)^.FileHeader.SizeOfOptionalHeader -
```

① 这些定义来自于 *WINNT.H*(Win32 SDK 中 WinNT 系统内核结构的，C 语法的声明文件)，可以在 *Windows.pas* 内找到 Win32 SDK 中绝大多数头文件转换成的 Pascal 声明。更多的内容可以从 JEDI API 中得到，它是一个开放源代码的项目(http://www.jedi-org.com/)。

```
  SizeOf(TImageOptionalHeader));

 // 节表
 inc(p, SizeOF(TImageNtHeaders));
 writeln('节表起始地址: $', IntToHex(Integer(p)-Integer(Buff), 8));

 // ...
```

在 PE 的可选头部中，有一个重要的数组："数据目录"。数据目录与节表的一部分信息是重叠的，因此可以把它看作是节表中一些重要数据的快速入口。数据目录仅仅是节表的一个子集。

下面的代码可以列出全部数据目录信息(部分代码参考上例)：

```
 // ...

 // PE 可选头部
{$RANGECHECKS OFF}
   writeln(' 数据目录信息 ==> ');
   writeln('Size':16, 'VirtualAddress':16);
   with PImageNtHeaders(p)^.OptionalHeader do
    for i := 0 to NumberOfRvaAndSizes - 1 do
    begin
     if DataDirectory[i].VirtualAddress = 0 then continue;
     system.write('--> ':8);
     system.write(IntToHex(DataDirectory[i].Size, 8));
     writeln(IntToHex(DataDirectory[i].VirtualAddress, 8):10);
    end;
```

数据目录通常由 16 个元素组成，以目前所知，前 15 个元素用作下面这些节的快速入口(如果相对应的节不存在，则该元素的 **VirtualAddress** 域为 0)：

```
IMAGE_DIRECTORY_ENTRY_EXPORT      = 0;   { Export Directory }
IMAGE_DIRECTORY_ENTRY_IMPORT      = 1;   { Import Directory }
IMAGE_DIRECTORY_ENTRY_RESOURCE    = 2;   { Resource Directory }
IMAGE_DIRECTORY_ENTRY_EXCEPTION   = 3;   { Exception Directory }
IMAGE_DIRECTORY_ENTRY_SECURITY    = 4;   { Security Directory }
IMAGE_DIRECTORY_ENTRY_BASERELOC   = 5;   { Base Relocation Table }
IMAGE_DIRECTORY_ENTRY_DEBUG       = 6;   { Debug Directory }
IMAGE_DIRECTORY_ENTRY_COPYRIGHT   = 7;   { Description String for x86}
IMAGE_DIRECTORY_ENTRY_ARCHITECTURE= 7;   { Arch. Specific Data for
                                           Alpahs }
IMAGE_DIRECTORY_ENTRY_GLOBALPTR   = 8;   { Machine Value (MIPS GP) for
                                           IA-64 }
IMAGE_DIRECTORY_ENTRY_TLS         = 9;   { TLS Directory }
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG= 10;   { Load Configuration Directory }
```

```
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT    =11;   { Bound Import Directory
                                               in headers }
IMAGE_DIRECTORY_ENTRY_IAT             =12;   { Import Address Table }
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT    =13;   { Delay Load Import
                                               Descriptors }
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR  =14;   { COM Runtime descriptor }
IMAGE_DIRECTORY_ENTRY_COMHEADER       =14;   { alias of COM_DESCRIPTOR }
```

需要注意的是，这个数据目录并不总是 16 个元素。编译器可以将它设为更高的值(既定数组元素的含义是不会因此而变化的)。然而，在 Delphi 中，使用定长数组来定义 DataDirectory 域，这可能导致直接使用 NumberOfRvaAndSizes 访问时超出数组边界，所以本例中使用了 {$RANGECHECKS OFF}编译条件。

同样的原因，不能直接使用 SizeOF(TImageNtHeaders)作为 PE 文件头的长度，因而本例中加入了修正可选头部长度的代码，以准确地定位节表。

6.1.2 节表

节表除了作为节的索引之外，还存放着节的一些信息，包括节的属性、大小等等。它是一个数组，数组长度由 PE 头中的 NumberOfSections 域指定，数组元素类型为 TImageSectionHeader。

用非常简单的代码就可以列举出全部的节表(部分代码参考上例)：

```
type
  TSectionHeaderArr = array [Word] of TImageSectionHeader;
  PSectionHeaderArr = ^TSectionHeaderArr;
  TSectionName = packed array[0..IMAGE_SIZEOF_SHORT_NAME-1] of char;

// ...
  tempSize := PImageNtHeaders(p)^.FileHeader.NumberOfSections;

  // 定位到节表
  inc(p, PImageNtHeaders(p)^.FileHeader.SizeOfOptionalHeader +
    SizeOf(TImageOptionalHeader));
  inc(p, SizeOF(TImageNtHeaders));
  writeln('VirtualAddress':32);
  for i := 0 to tempSize - 1 do
    with PSectionHeaderArr(p)^[i] do
      Writeln(TSectionName(Name):13, ' --> ', IntToHex(VirtualAddress, 8));
// ...
```

图 6.3 是使用示例程序列出的 *Delphi32.exe* 的数据目录和节表信息。通过对比可以清楚地

看到数据目录与节表之间的关系：数据目录是节表的一个子集，用于快速定位节。

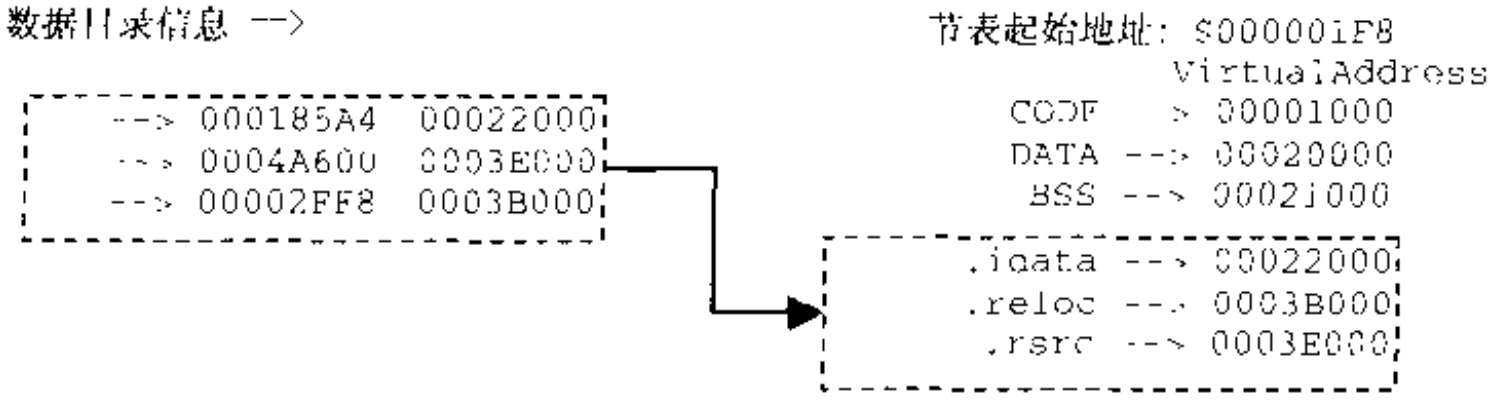


图 6.3　数据目录与节表之间的关系(一)

但是，不要被这个图迷惑：它并不表明在节表中一定存在一个项与数据目录对应。相反，一些PE 文件的数据目录(数组)有效元素数目会多于节表(数组)元素数目(也存在前者少于后者的情况)，因为编译器或者优化工具合并了一些节，使得数据目录中的多个元素被包含在一个节中。同样地，列出 *Explorer.exe*，可能如图 6.4 所示。

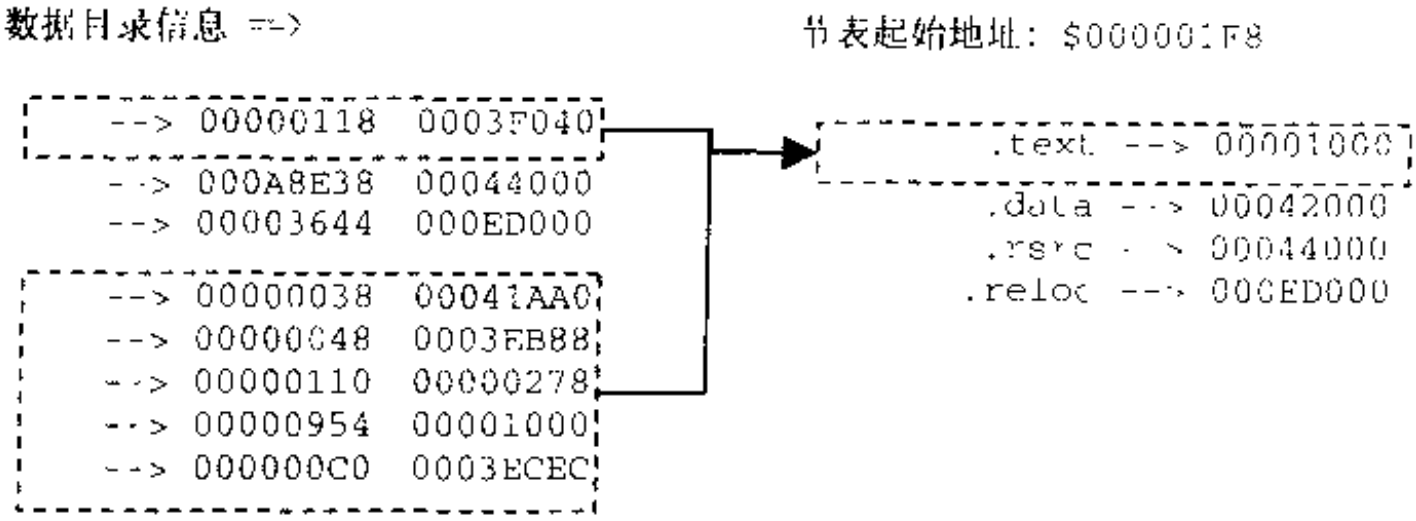


图 6.4　数据目录与节表之间的关系(二)

6.1.3　节

紧接在节表之后的就是节。但节是不定长的，除了从节表或数据目录快速地索引到节，其他的方法都会存在不兼容的问题。节表的元素数与节的数目是不一定相等的，因为有些节可能不占物理空间，例如后面将说到的 **BSS** 节。

节的名字与属性等信息都存放在节表中，节的内容是一些既定格式或由编译器约定格式的二进制数据。更进一步地讲，可以是任何数据。通常而言，具有如下名称的节是具有既定格式的(见表 6-1)。

表 6-1 具有既定格式的节

名称	内 容	入口[1]	结 构	Delphi
.text .code	可执行代码			CODE
.bss	未经初始化的数据			BSS
.data	已经初始化的数据			DATA
.edata	导出文件名表	0	IMAGE_EXPORT_DIRECTORY	.edata
.idata	导入文件名表	1, 11, 12[2]	IMAGE_THUNK_DATA IMAGE_IMPORT_BY_NAME IMAGE_IMPORT_DESCRIPTOR IMAGE_BOUND_IMPORT_DESCRIPTOR IMAGE_BOUND_FORWARDER_REF	.idata
.rdata	调试目录、只读数据	6	IMAGE_DEBUG_DIRECTORY	.rdata
.debug	调试信息		IMAGE_DEBUG_MISC IMAGE_DEBUG_TYPE_MISC IMAGE_SEPARATE_DEBUG_HEADER	.debug
.reloc	重定位表信息	5	IMAGE_RELOCATION IMAGE_BASE_RELOCATION	.reloc
.rsrc	资源	2	IMAGE_RESOURCE_DIRECTORY //Resource Format...	.rsrc
.tls	线程的本地存储器	9	IMAGE_TLS_DIRECTORY	.tls
.xdata	异常句柄表	3	IMAGE_RUNTIME_FUNCTION_ENTRY	
.descr	版权信息	7		
.CRT	C++运行期数据			
	安全目录	4	WIN_CERTIFICATE	
.didat	延迟导入文件名表	13	ImgDelayDescr	
	映像的加载配置	10	IMAGE_LOAD_CONFIG_DIRECTORY	

① 参见此前“文件头”小节中对“数据目录(DIRECTORY_ENTRY)”的描述。

② 数据目录(数组)的第 11,12 两个元素分别是 BOUND_IMPORT 和 IAT。尽管它们被列在这里，但只是说明这些数据与导入表有关，并不表明它们一定在该节中。例如在 VC 编译的 WDM 驱动(.SYS)文件中，IAT 表指向.rdata 节；在另一些可执行文件中，也可能指向.text 节。

续表

名称	内　容	入口	结　构	Delphi
.arch[①]	平台(体系)描述	7	IMAGE_ARCHITECTURE_HEADER	
[②]	机器值(MIPS GP)	8		
[③]	COM+ 2.0运行期描述	14	IMAGE_COR20_HEADER // metadata or method IL // for .NET...	
.pdata[④]	异常信息		Only for .OBJ...	
.debugSS	Codeview格式的符号记录			
.debugST	Codeview格式的类型记录			
.debugSP	预编译头信息			
.drectve	为Linker传入的指示信息			

节的命名只是惯例，而不是绝对的。可以使用任意名字来命名节，并使它具有与名字完全无关的性能——操作系统的文件装载器是根据节的属性而不是名字来识别和装载节的。

特定的编译器生成的文件通常具有固定的节名。Delphi 编译的程序更是如此。如果需要自由地指定编译后 PE 文件节的名字与格式，应该选择汇编语言而不是像 Delphi 这样的高级语言。

无论如何，在数据目录中指定的节总是具有既定含义的，这也是操作系统与编译器之间相互遵守的约定。

6:1.4　PE 文件与内存映射

PE 文件的物理存储格式与它在内存中的布局存在对应关系，但从地址上来讲，这种关系并不是绝对的。

具体来说，DOS 文件头、PE 文件头和节表都会被直接映射到内存中，但节会按照它在节表中的属性设定映射。这可能使得内存中的节并不像它在文件中一样是连续的，且节的空间占用

① 在 Alpha 平台上，这个目录项被用作平台描述；而在 x86 上，(如果被使用的话，)通常用作版权信息描述。
② 通常这个数据目录(GLOBALPTR)入口在 x86 的平台上未用，而是用于 IA-64 平台。
③ 通常这个数据目录(COM_DESCRIPTOR)入口指向 *.text* 节。
④ *.pdata* 有时也和 *.xdata* 一样作为异常句柄表使用，例如在一些 *DLL* 文件中。

的大小也可能会发生变化，如图 6.5 所示。

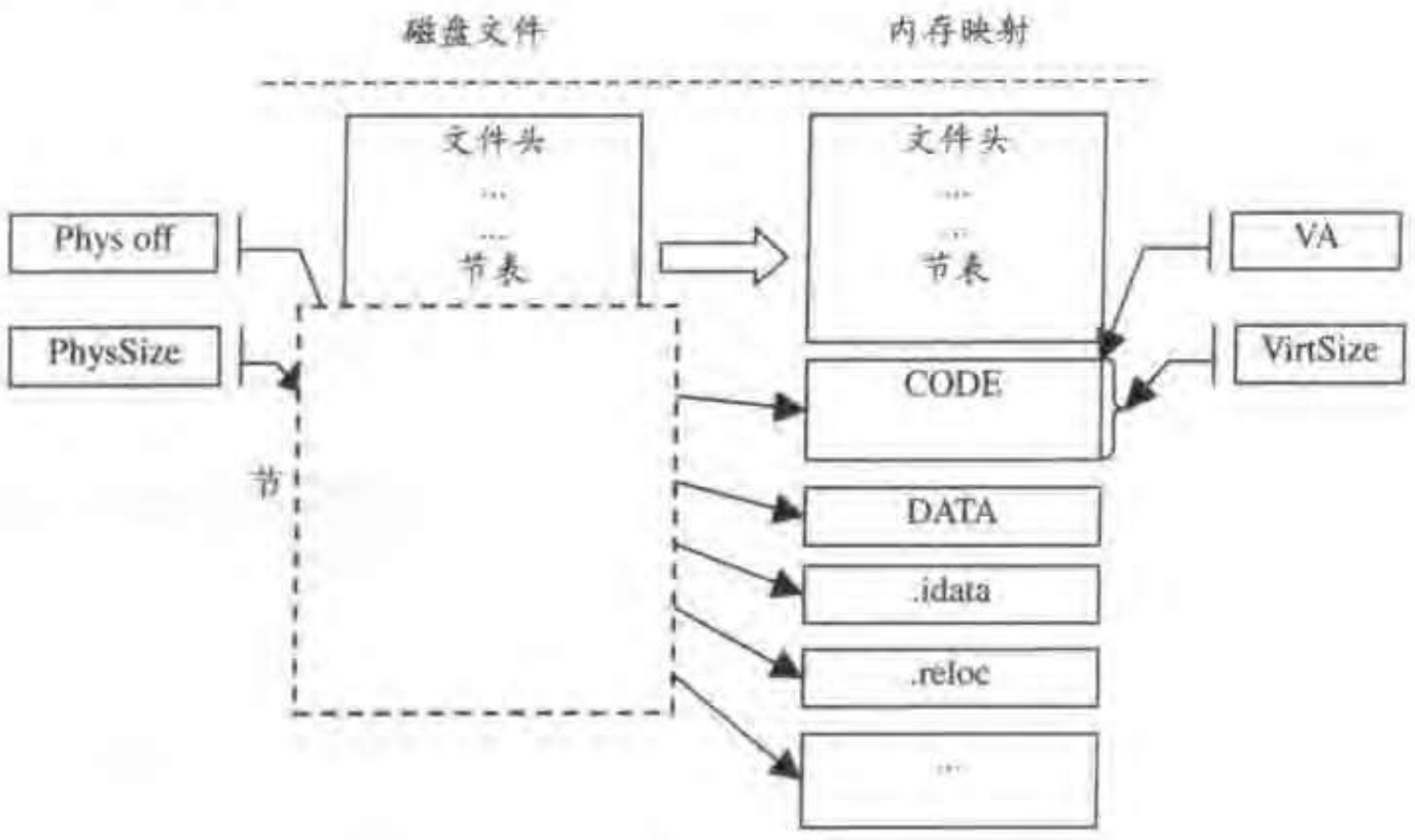


图 6.5 文件与内存映像的映射关系

以 *Delphi32.exe* 为例，用 TDump 查看它的节表：

```
C:\>tdump E:\Delphi7\Bin\delphi32.exe
#   Name      VirtSize   RVA      PhysSize  Phys off  Flags
--  --------  ---------  -------- --------  --------  --------
01  CODE      0001EEF4   00001000 0001F000  00000400  60000020 [CER]
```

从这部分数据中可以看到：CODE 节在物理文件中占$1F0000 个字节，但在内存映像中只占$1EEF4 字节[①]。它在物理文件中起始位置相对于头件头的偏移$400 处，在映射到内存后，则位于模块起始地址的相对偏移$1000 处。

为了描述 PE 文件的节和数据在不同映射关系下的地址，PE 格式中引入了三套不同的地址系统。如表 6-2 与表 6-3 所所示。

① 实际情况并非如此，PE 文件中关于节对齐的设定将会影响到节在内存中的实际占用大小，参见本章第 3 节"Delphi 的 PE 文件中的一些重要节"。

表 6-2　不同地址系统的使用范围

简称	含　义	应用范围	注　释
RAW Address	文件物理地址	文件编辑；十六进制编辑器，资源修改器等	Phys Address
RVA	相对虚地址	PE文件；用于记录地址数据	参见下表
VA	内存中的虚地址	在运行期的内存映像中的地址 可在调试工具或者异常转储工具中看到	

表 6-3　各种 RVA 的使用范围[①]

简称	基　点	应　用	备注
hRVA	PE文件头中的设定值	PE文件头部的一些结构的域值(数据)	
tRVA	节表中的设定值	操作系统将PE文件中的节映射到内存时，用这些信息来定位	
sRVA	节映射的逻辑起点	部分节中的结构的域值(数据)，编译器用来定位运行期数据、资源等	
bRVA	ImageBase	通常被编译器或反编译器使用的一种地址	②

首先是磁盘文件上的物理地址，通常以 RAW Address 或 Phys Address 来表示。显然，总的地址空间等于文件大小。物理地址通常在对文件进行十六进制编码时用到。

其次是在 PE 文件头部、节表和节中使用的相对虚地址，通常以 RVA(Relative Virtual Address)来表示。根据参照的基点的不同，RVA 可以分成很多种。在需要区分几种 RVA 时，可以用 hRVA(头部)、tRVA(节表)、sRVA(节)等来指代。

第三种是 PE 文件被载入内存后的地址。由于在 Win32 环境下使用的是 4G 的虚拟内存(Virtual Memory)，所以这个地址被称做虚地址 VA (Virtual Address)。这个地址以一个映像在内存中的实际载入地址为基点(基地址)，这是它与相对虚地址(这里指的是 bRVA)的惟一不同。虚地址是程序运行期实际使用的地址，通常可以在调试器或者运行代码的转储文件中看到。

如果文件载入到内存中时与编译器假定的内存基点地址正好一致(相对基地址=基地址)，那么一些相对虚地址值(RVA)与虚地址值(VA)是一样的。这种情况一般发生在.EXE 文件上。这是因为.EXE 载入时，进程空间通常比较空闲。

① 将 RVA 细节成多种类型，并不是习惯性的用法。本书中较少的一些地方会使用到。

② 亦即是通常所说的 RVA。它以 PE 文件载入到一个假定的内存地址为基点，这个假定的内存地址由编译器写在 PE 文件头结构中的 ImageBase 域，通常用于静态反编译。编译器编码时使用的内存寻址都是基于 bRVA 的(例如代码节)，在操作系统中通过重定位操作，最后将内存映像的数据转换为 VA。有关重定位的细节，可以参阅本章的 6.3 节。

但对于DLL来说，编译器通常设定多个DLL为相同的相对基地址。如果这些DLL又需要在同一个进程中载入，就会存在地址冲突。在这种情况下，操作系统的载入例程就会以一个不可预知的虚地址来载入DLL，因此相对基地址就与基地址不同。

下面的例程PosToSectionEntry()用于返回数据目录中的节的相关信息：

```
// BASE = 文件映像在内存中的起始地址
// Section = [0,.IMAGE_DIRECTORY_ENTRY_IAT]
// sRVA = 节的VirtualAddress(RVA)
// RawAddr = 节的物理地址(Phys Off)，相对于BASE的偏移
function PosToSectionEntry(BASE: Pointer; Section: SmallInt;
   Var sRVA, RawAddr: DWORD): Boolean;
var
 p : pByte;
begin
 RawAddr := 0;
 Result := False;

 // 定位到PE头，取导入表的VirtualAddress
 p := BASE;
 inc(p, PImageDosHeader(p)^._lfanew);
 with PImageNtHeaders(p)^ do
 begin
  tempSize := FileHeader.NumberOfSections;
  sRVA := OptionalHeader.DataDirectory[Section].VirtualAddress;
  if sRVA = 0 then
   Exit;
end;

// 定位到节表数组，取导入表的物理地址
inc(p, PImageNtHeaders(p)^.FileHeader.SizeOfOptionalHeader -
 SizeOf(TImageOptionalHeader));
inc(p, SizeOf(TImageNtHeaders));
for i := 0 to tempSize - 1 do
 with PSectionHeaderArr(p)^[i] do
  // 如果虚地址相等或者sRVA落在节的内部
  if (sRVA = VirtualAddress) or
   ((sRVA > VirtualAddress) and
   (sRVA < VirtualAddress + Misc.VirtualSize)) then
  begin
   RawAddr := PointerToRawData + (sRVA - VirtualAddress);
```

```
        Result := True;
        Break;
      end;
end;
```

6.1.5　有关相对虚地址的计算

按照概念来讲，虚地址到物理地址的计算公式是这样：

```
physAddr = 相对基点的物理地址 + RVA值相对于基点的偏移
```

相应的，也可以得出一个与上面公式相类同的 VA 计算公式：

```
VA = 相对基点的虚地址 + RVA值相对于基点的偏移
```

不过这个 VA 计算公式比较不常用。因为计算 VA 时有一个公式更加简单：

```
VA = ImageBase + RVA值
```

在这个公式里，不需要考虑相对虚地址的不同类型(例如 tRVA 或 hRVA)。因此大多数情况下，区分 RVA 类型的目的，是为了便于在十六进制编辑时计算物理地址。

下面列举各种地址之间的转换计算方法。在计算之前，(若未明确指出，则)先假定被载入文件是 Delphi32.exe，文件载入到内存地址$00400000。

■ hRVA 的计算

在文件头中，DOS 文件头的_lfanew 域值的基点是文件头部，这是一个 hRVA 类型的相对虚地址值。由于操作系统将该文件载入到$00400000，则文件头部的 VA 值即是$0040000。计算_lfanew 域的 VA 值的过程如下：

```
VA  = ImageBase + RVA值
    = $00400000 + $00000100
    = $00400100
```

由于文件头部被直接映射到内存。因此它的 hRVA 值其实也是实际物理偏移值。这表明_lfanew 域所指向的值(PE 文件头)的物理地址为：

```
physAddr = 相对基点的物理地址 + RVA值相对于基点的偏移      +
         = 0 + hRVA
         = 0 + $00000100
         = $100
```

■ tRVA 的计算

从文件头开始，可选头部(含数据目录)和节表都是直接映射到内存中，因此它们的 hRVA 值与实际物理值是相等(参见上面对 hRVA 的计算)。然而接下来的各个节却不是这样，节按照节表的设定值来建立映射关系。

建立这种映射关系时使用的值被称为 tRVA，这些值存在于节表中。tRVA 通常用作其它计算时的参照值，这在后面的 sRVA 计算中会用到。用 TDUMP 来查看 *Delphi32.exe* 的节表：

```
C:\>tdump E:\Delphi7\Bin\delphi32.exe
#  Name     VirtSize   RVA      PhysSize  Phys off  Flags
-- -------- ---------- -------- --------- --------- --------
01 CODE     0001AEF4   00001000 0001B000  00000400  60000020 [CER]
02 DATA     00000308   00020000 00000400  0001B400  C0000040 [IRW]
03 BSS      000001CD   00021000 00000000  0001B800  C0000000 [RW]
04 .idata   00018584   00022000 00018600  0001B800  C0000040 [IRW]
05 .reloc   00002FF8   0003B000 00003000  00037E00  50000040 [IRS]
06 .rsrc    0004A600   0003E000 0004A600  0003AE00  50000040 [IRS]
```

以 *.idata* 节为例，计算该节的 VA 值的过程如下：

```
VA  = ImageBase + RVA值
    = $00400000 + $00022000
    = $00422000
```

根据节表中的值，该节的物理地址为$0001F800。

尽管数据目录中也存在着 tRVA，例如在数据目录中导入表(IMAGE_DIRECTORY_ENTRY_IMPORT)的 RVA 值也是$00022000。然而数据目录中没有相应的物理地址数据。因此在计算物理地址时，数据目录起不到太多的作用。

■ sRVA 的计算

在节中，一些数据结构的域值使用相对地址，但相对的基点是该节的 tRVA。以导入表中的第一个导入模块的描述(_IMAGE_IMPORT_DESCRIPTOR)为例，该结构的 Name 域值为 $00026A2C，则计算它的 VA 值的过程如下：

```
VA  = ImageBase + RVA值
    = $00400000 + $00026A2C
    = $00426A2C
```

而计算物理地址时却复杂得多，因为这需要通过先计算 tRVA 的物理地址。仍以 Name 域

为例，计算的过程如下：

```
physAddr = 相对基点的物理地址 + RVA值相对于基点的偏移
         = 相对基点(tRVA)的物理地址 + (sRVA - 相对基点的tRVA)
         = $0001F800 + ($00026A2C - $00022000)
         = $24A2C
```

这样在物理文件的$2422C 偏移位置就可以找到该模块的名字：*rtl70.bpl*。

这个例子相对简单，因为这里的“相对基点的物理地址”是在上面的“tRVA 的计算”中通过查节表得到的。然而由于数据目录与节表并没有一一对应关系，所以要以数据目录中的 tRVA 值为基点，来计算一些 sRVA 的物理地址，则更为复杂。例如 *CabArc.exe* 这个程序中的数据目录中存在一个 IAT 表，其入口 RVA 值为$0003A11C：

```
C:\>tdump E:\Delphi7\Bin\cabarc.exe | find /i "Import Addr Table"
Name                 RVA      Size
-------------------  --------  --------
Import Addr Table    0003A11C  000000F4
```

CabArc.exe 的节表如下：

```
C:\>tdump E:\Delphi7\Bin\cabarc.exe
#  Name     VirtSize  RVA      PhysSize  Phys off  Flags
-- -------- --------  -------- --------  --------  --------
01 .text    0000D29F  00001000 0000D400  00000400  60000020 [CER]
02 .rdata   000003B7  0000F000 00000400  0000D800  40000040 [IR]
03 .data    000291C4  00010000 00002000  0000DC00  C0000040 [IRW]
04 .idata   00000658  0003A000 00000800  0000FC00  C0000040 [IRW]
05 .reloc   00000E44  0003B000 00001000  00010400  42000040 [DIR]
```

对照节表可知 IAT 表的 RVA 指向.idata 节中的数据，因此其相对基点是.idata 节。在程序中，这个“对照”的过程可以参考前面提供的 PosToSectionEntry()例程，该例程将遍历节表并为指定的数据目录项返回参考基点的 sRVA 和物理地址[①]。

在 *CabArc.exe* 的 IAT 表中第一个项(_IMAGE_THUNK_DATA)的 AddressOfData 域值为 $0003A42C，则相应的物理地址的计算过程为：

```
phyAddr = 相对基点(tRVA)的物理地址 + (sRVA - 相对基点的tRVA)
        = .idata表的物理地址 + (sRVA - .idata表的tRVA)
        = $0000FC00 + ($0003A42C - $0003A000)
```

① 该例程并不返回作为参考基点的节的 tRVA，而是直接返回经过计算的 sRVA 和 physAddr 值。如果需要 tRVA，则他们的遍历过程是一致的。

```
        = $1002C
```

在物理文件的$1002C偏移位置可以找到一个PIMAGE_IMPORT_BY_NAME的结构，表明这个AddressOfData域指向一个名为LCMapStringA的导入项。不过需要注意的是，在运行期，内存中的IAT表将直接接向被导入例程的入口地址，而非PIMAGE_IMPORT_BY_NAME结构。也就是说，在运行期映像中应当使用_IMAGE_THUNK_DATA.Function_域值。

■ bRVA的计算

bRVA是实际地址值而非偏移值，因此VA值通常不需要计算。但如果相对基点发生了变化，则是例外。例如重定位表项是基于ImageBase的bRVA值，如果文件载入到内存时没有使用ImageBase，那么就需要重定位计算。这时：

```
VA = bRVA + (New_ImageBase - ImageBase)
```

关于重定位项的查找和VA计算的细节，可以参见本章第3节的“重定位节(.reloc)”小节。

bRVA到物理地址的计算，需要先将bRVA转换成sRVA，此后的计算就与sRVA一致了：

```
sRVA = bRVA - ImageBase
```

仍以重定位项为例。例如*Delphi32.exe*在CODE节中存在一个重定位项，其RVA是$1FE84，原始值为$41EF1C，是*Delphi32.exe*的模块初始化代码中传入的单元初始化表地址。这个值$41EF1C就是bRVA值。因此这个单元初始化表的物理地址的过程为：

```
sRVA     = bRVA - ImageBase
         = $0041EF1C - $00400000
         = $1EF1C

phyAddr = 相对基点(sRVA)的物理地址 + (sRVA - 相对基点的sRVA)
        = CODE表的物理地址 + (sRVA - CODE表的sRVA)
        = $00000400 + ($0001EF1C - $00001000)
        = $1E31C
```

在这个例子中，值$1EF1C所相对的基点是CODE节，这也是通过遍历节表来查找得到的。其细节可以参考上一小节“sRVA的计算”中的描述。

6.2　Delphi 的 PE 文件头中一些重要的域

Delphi 的编译器生成的 PE 文件的头部都是类同的，在用户代码中无法自定义 PE 头部。可以使用第三方工具修改编译器生成后的文件，这些工具包括：PEditor、LordPE、stud_PE 等。部分 PE 结构(例如资源)需要专用的编辑器。

操作系统根据 PE 文件头中的信息来决定是否装载文件，以及需要以怎样的方式来装载和映射文件到内存中。这些信息通常由编译器直接写在文件头中。表 6-4 是 Delphi 生成的 PE 文件的一些重要且固定的属性标志。

表 6-4　Delphi PE 头的重要域

结　构	域	值	含　义
PE头部	Machine	$014C	支持Inter I386以上芯片
	PointerToSymbolTable	$0	未用 Delphi使用专门的节来存放调试时用的符号信息
	NumberOfSymbols	$0	
	SizeOfOptionalHeader	$00E0	Delphi编译的PE文件的可选头部定长
	Characteristics	$818E	参见后续表
		$A18E	
可选头部	Magic	$10B	表示正常的(一般性的)EXE Image
	ImageBase	$00400000	默认载入的基地址，可在编译选项中设定
	SectionAlignment	$00001000	节对齐 影响操作系统映射节到内存的方式
	FileAlignment	$00000200	文件对齐 影响编译器生成PE的节时的虚地址计算
	MajorOperatingSystemVersion	$0004	所需系统的(最低)版本：4.0
	MinorOperatingSystemVersion	$0000	
	MajorImageVersion	$0000	自定义映像版本：未用
	MinorImageVersion	$0000	
	MajorSubsystemVersion	$0004	所需的子系统(最低)版本：4.0
	MinorSubsystemVersion	$0000	
	Win32VersionValue	$00000000	Win32版本值：未用

续表

结 构	域	值	含 义
可选头部	SizeOfHeaders	$00000400	Delphi生成的PE头部是定长的
	CheckSum	$00000000	总是为0值(操作系统只验证VDM驱动的CheckSum的有效性)
	Subsystem	$0003	控制台模式的.EXE
		$0002	GUI模式的.EXE、.DLL和.BPL等
	DllCharacteristics	$0000	未用
	SizeOfStackReserve	$00100000	栈的保留空间和初始交付空间的大小
	SizeOfStackCommit	$00004000	EXE模块可在编译选项中设定这些值；包和DLL 使用HOST的栈空间，此值无意义，通常为0。
	SizeOfHeapReserve	$00100000	堆的保留空间和初始交付空间的大小
	SizeOfHeapCommit	$00001000	
	LoaderFlags	$00000000	未用

表 6-5 是 Characteristics 域的详细设定。

表 6-5 Delphi PE 头的重要域-Characteristics

Characteristics域			.EXE	.DLL和.BPL
位设定	值	含义		
IMAGE_FILE_EXECUTABLE_IMAGE	$0002	可执行	Y	
IMAGE_FILE_LINE_NUMS_STRIPPED	$0004	无行号信息	Y	
IMAGE_FILE_LOCAL_SYMS_STRIPPED	$0008	无本地符号信息	Y	
IMAGE_FILE_32BIT_MACHINE	$0100	需要32位主机	Y	
IMAGE_FILE_BYTES_REVERSED_LO	$0080	保留	Y	
IMAGE_FILE_BYTES_REVERSED_HI	$8000			
IMAGE_FILE_DLL	$2000	动态链接库	N	Y
		(实际值)	$818E	$A18E

6.3　Delphi 的 PE 文件中一些重要的节

缺省情况下，Delphi 的编译器生成的 PE 文件具有如下的节。这些节固定的属性标志(Flags)如下：

```
C:\>tdump E:\Delphi7\Bin\serverinfo.exe
#   Name     VirtSize   RVA      PhysSize  Phys off  Flags
--  -------- ---------- -------- --------- --------- --------
01  CODE     0005CEA0   00001000 00050000  00000400  60000020 [CRF]
02  DATA     00001F10   0005E000 00002000  00050400  C0000040 [IRW]
03  BSS      00000C69   00060000 00000000  00052400  C0000000 [RW]
04  .idata   000037A0   00061000 00003800  00052400  C0000040 [IRW]
05  .tls     00000018   00065000 00000000  00055C00  C0000000 [RW]
06  .rdata   00000018   00066000 00000200  00055C00  50000040 [IRS]
07  .reloc   00007904   00067000 00007A00  00055E00  50000040 [IRS]
08  .rsrc    00017800   0006F000 00017800  0005D800  50000040 [IRS]
```

此外，编译一个.DLL 或包时，通常会生成一个.edata 节，除非该.DLL 没有例程导出。如下所示：

```
C:\>tdump E:\Delphi7\Bin\exptdemo.dll
#   Name     VirtSize   RVA      PhysSize  Phys off  Flags
--  -------- ---------- -------- --------- --------- --------
05  .edata   00000043   0007F000 00000200  0007C000  50000040 [IRS]
```

如果编译时选择了“Include TD32 debug info”，则会生成一个.debug 节，用来存储调试信息。如下所示：

```
C:\>tdump E:\Delphi7\Bin\tregsvr.exe
#   Name     VirtSize   RVA      PhysSize  Phys off  Flags
--  -------- ---------- -------- --------- --------- --------
09  .debug   000E1AAC   0001C000 000E1AAC  00017200  50000040 [IRS]
```

除了 CODE、DATA、BSS、.idata、.reloc、.rsrc 这几个节总是存在以外(即使是空的)，Delphi 的编译器不会将一个无用的、空的节放在 PE 文件头中。

6.3.1　线程局部存储(.tls 和.rdata)

.tls 用来存放线程局部变量的一些初始化信息，而.rdata 用来存放其目录。

.tls 与.rdata 节缺省存在的原因是：在 *system.pas* 和 *sysinit.pas* 内置了线程局部存储管理

器。因此，如果使用“带运行期包”的选项编译，那么.tls 节和.rdata 节将不存在——*rtl.bpl* 中已经有了这些线程局部存储的一份拷贝——但是，如果在模块中定义了 threadvar，则这两个节又会出现，因为编译器必须为当前模块初始化一个线程局部存储。

在最小化内核中，因为没有线程局部存储管理器，所以也不会有这两个节。

另外一种情况也会导致.rdata 节的出现：使用{$DESCRIPTION}定义了模块的描述字符串，但.tls 节不会因此而出现。

.rdata 也会在其他的一些情况下出现，例如 OLE 开发中使用的 GUID 也用该节来存储。

6.3.2 资源节(.rsrc)

资源是 PE 结构中讨论得最多的数据之一。但资源并不复杂，所有的资源都使用比较一致的数据结构来描述。这些定义并没有出现在 *Windows.pas* 中，但可以在 Jedi Api 中查找到绝大多数的定义。下面使用 LordPE 分析某一个 *.EXE* 时的资源 dump 树来解析资源的结构(仅保留了一些重要的域)：

```
->Resource Tree (detailed dump)
  ( 资源从一个TImageResourceDirectory结构开始 )
  [Resource Directory (0)]:
  NumberOfNamedEntries:  0x0000
  NumberOfIdEntries:     0x0002
  ------------------------------------------------------------
  (在TImageResourceDirectory结构后紧接着一个TImageResourceDirectoryEntry为
   基类型的数组，数组长度 = NumberOfNamedEntries + NumberOfIdEntries)
  [ResourceEntry]:
  Name/Id:    0x00000006        ( STRING TABLE，仅有根目录中的ID是用来表示资
                                源类别的 )
   OffsetToData:  0x80000020    ( 高位为1，DATA_IS_DIRECTORY，相对于节起点的
                                偏移是$20 )
     [Resource Directory (1)]:      ( Address => 节起点 + $20 )
     NumberOfNamedEntries:  0x0000
     NumberOfIdEntries:     0x0007
       [ResourceEntry]:
       Name/Id:       0x00000F9A    ( 高位为0，名字是一个ID )
       OffsetToData:  0x80000088    ( 高位为1，DATA_IS_DIRECTORY，相对于节起点
                                    的偏移为$88 )
         [Resource Directory (2)]: ( Address => 节起点 + $88 )
         NumberOfNamedEntries:  0x0000
         NumberOfIdEntries:     0x0001
           [ResourceEntry]:
```

```
        Name/Id:        0x00000000  ( id=0, 一个资源的数据块虽然采用的结构有
                                      Name域, 但真实名字却在上一级 )
      OffsetToData:     0x00000160  ( 高位为0, 则表明指向真实的数据块. 其偏移
                                      为$160, 数据类型为 TImageResourceDataEntry )
        [ResourceDataEntry]:  ( Address => 节起点 + $160 )
        OffsetToData (RVA):   0x0001B218 ( 只有资源的数据块使用了sRVA, 而
                                          结构中的其他定位使用的都是偏移 )
        Size:               0x00000184
        CodePage:           0x00000000
        Reserved:           0x00000000

    [ResourceEntry]:
    // ...
    // 剩下6个资源目录的入口
-------------------------------------------------------------------
  [ResourceEntry]:
  // ...
  // 根目录中剩下的一个资源目录的入口
```

随书光盘的示例程序 PEHeader.dpr 中的 EnumResources()例程更详细地演示了如何遍历资源的结构。例程中取资源数据块时，使用的是从 sRVA 转换为偏移的方法。注意该例程只演示了如何从字符串表(String Table)中取一个字符串，而不是全部。

Delphi 中有一套自己的资源管理和调用机制，这一部分内容可以参见第 10 章“资源的管理和使用”。

6.3.3　导入、导出表(.idata 和.edata)

PE 结构中的导入表(.idata)，用来保存模块引用的外部例程的列表及相关信息。事实上，无论是否引用外部例程，Delphi 的编译器都将为 PE 文件建立一个.idata 节[①]。

导出表(.edata)用来向其他模块公开自己包含的可供调用的例程列表以及接口信息。Delphi 的包也用导出表来导出对象和一些内部数据结构。

通常.edata 节会出现在.dll 和.bpl 模块中，但是.exe 中也可以有导出表。

① 可以用 Delphi 编写一个不使用任何导入例程的可执行程序，但是通过 tdump 会发现，这个程序仍然会有.idata 节。

■ .idata 的结构及使用

导入表是一个 TImageImportDecriptor 类型的数组。它的 Name 域是一个 DWORD 类型的 sRVA 值，如果值为 0，则表明数组结束[①]。下面的代码列出模块所有导入的其他模块的名字：

```
// uses unit JwaWinNT.pas from Jedi API
var
  D: PByte;
  Buff: Pointer;
  i, tempSize : Integer;
  pBASE, VABase, RawAddr, sRVA : DWORD;
begin
  // ...
  // 将文件读到Buff所指向的内存块

  // 定位到导入表，输出导入模块
  p := Buff;
  if PosToSectionEntry(p, IMAGE_DIRECTORY_ENTRY_IMPORT,
    VABase, RawAddr) then
  begin
    writeln('导入表起始地址: ', IntToHex(RawAddr, 8));

    inc(p, RawAddr);
    pBASE := DWORD(p);
    while PImageImportDecriptor(p)^.Name <> 0 do
      with PImageImportDecriptor(p)^ do
      begin
        if Union.OriginalFirstThunk <> 0 then
          writeln(pChar(pBASE+(Name-VABase)):16,
            IntToHex(Union.OriginalFirstThunk, 8):10)
        else
          writeln(pChar(pBASE+(Name-VABase)):16,
            IntToHex(FirstThunk, 8):10);
        inc(p, SizeOf(TImageImportDecriptor));
      end;

// ...
```

从 TImageImportDecriptor 结构中只能列举到被导入模块的信息，如果想知道导入的例程列表，则要对它的 Union.OriginalFirstThunk 和 FirstThunk 域做进一步地探索：

① 事实上，导入表数组的最后一个元素为空值，也就是说，该元素所有的域都为 0。在 PE 结构中，有许多数组是没有存储 Length 值的，而是采用“数组的最后一个元素为空”作为数组的结束标志。

```
// ...
// 取得一个导入描述符到指针p，类型为PImageImportDecriptor

if PImageImportDecriptor(p)^.Union.OriginalFirstThunk <> 0 then
  sRVA := PImageImportDecriptor(p)^.Union.OriginalFirstThunk;
else
  sRVA := PImageImportDecriptor(p)^.FirstThunk;
DWORD(p) := pBASE + (sRVA - VABase);

// 以下开始列举导入例程列表
while PImageThunkData(p)^.Function_ <> 0 do
begin
  if PImageThunkData(p)^.Ordinal and IMAGE_ORDINAL_FLAG32 <> 0 then
  // 高位为1
    writeln(format('ord: %17.8x', [PImageThunkData(p)^.Ordinal and
      $7FFFFFFF]))
  else
  begin
    sRVA := PImageThunkData(p)^.AddressOfData;
    with PImageImportByName(pBASE + (sRVA - VABase))^ do
      writeln(format('name: %-16s, Hint: %.4x',
        [string(pChar(@Name)), Hint]))
  end;
  inc(p, SizeOf(TImageThunkData));
end;
```

在TImageThunkData记录的域中，Function_域保存了导入例程在目标模块中的sRVA。但是，这只是物理文件中的状况，对于一个已经载入到内存中的模块来说，操作系统的载入例程会将 Function_填成被导入模块中指定例程的入口地址。要获取一个已载入模块的原始Function_值，应当访问TImageImportDecriptor的Union.OriginalFirstThunk域。

OriginalFirstThunk 和 FirstThunk 被设计成指向两个完全等值的TImageThunkData 类型的数组。这样做的目的，是使得在不同的时期，PE 头中总保存着一份 TImageImportByName 数组，使得载入例程或应用程序可以查找到导入例程的名字(或序号)列表。然而这只是理想的状况，因为并非所有的编译器都会按约定在 OriginalFirstThunk 保存一个备份数组[①]。——至少 Delphi 就不会。因此，依赖 OriginalFirstThunk 在运行期查找导入名字的方法并不可靠。在这种情况下，只能根据 TImageThunkData 中存放的Function_值，在被导入模块的导出表中去查找例程的名字。

TImageThunkData 其实是一个 DWORD 类型的值，而非记录。但是 Delphi 使用变体记录

① 一些情况下，某些编译器生成的 PE 文件 FirstThunk 也不会保存另一个备份数组，这使得处理时变得复杂。

来定义它的数据结构，这使得 TImageThunkData 在不同的时期(物理文件、运行期映像等)可以被理解成不同的值。要想更好地理解这种结构，下面的代码将是一个不错的参考：

```
var
  ThunkData : DWORD;
  isFileMap : Boolean = True;

type
  TThunkDataArr = array [0..0] of DWORD;
  PThunkDataArr = ^TThunkDataArr;

    // ...
    // (参见上例. p指向 TImageThunkData 数组的头部)
{$RANGECHECKS OFF}
    // 以下开始列举导入例程列表
    for i := 0 to MaxInt do
    begin
      ThunkData := PThunkDataArr(p)^[i];
      if ThunkData = 0 then
        Break;

    if isFileMap then
      if ThunkData and IMAGE_ORDINAL_FLAG32 <> 0 then
        // TImageThunkData.Ordinal
        writeln('高位为1, 导入时使用序号: ', ThunkData and $7FFFFFFF)
      else
      begin
        sRVA := ThunkData;
        // TImageThunkData.AddressOfData
        with PImageImportByName(pBASE + (sRVA - VABase))^ do
          writeln('高位为0, 导入时使用名字: ', pChar(@Name));
      end
    else  // is Runtime MemoryMirror
      // TImageThunkData.Function_
      writeln('运行期, 载入例程修改该域为导入例程入口地址: $',
        IntToHex(ThunkData, 8));
  end;
// ...
```

■ .edata 的结构

导出表的结构在 *Window.pas* 中定义为 TImageExportDirectory。主要的三个域如下：

```
  AddressOfFunctions: ^PDWORD;
  AddressOfNames: ^PDWORD;
```

```
    AddressOfNameOrdinals: ^PWord;
```

分别指向三个存放导出例程的名字、RVA 地址和序号的数组。

AddressOfFunctions 数组的长度值保存在 NumberOfFunctions 中。而其他两个数组是等长的，长度值保存在 NumberOfNames 中。AddressOfNames 存放导出例程名字的 PChar 类型的指针，在与它对应的 AddressOfNameOrdinals 中，存放例程在 AddressOfFunctions 数组中的下标值。

在一个应用程序中使用序号导入例程时，使用的并不是AddressOfFunctions的下标值。序号减去 TImageExportDirectory.Base 的值，才是真正的下标。

下面的代码简单地列出了模块中使用名字导出的例程列表：

```
type
  TFunctionRVA = array [0..0] of DWORD;

var
  p: PByte;
  Buff: Pointer;
  i, tempSize : Integer;
  pBASE, VABase, RawAddr, sRVA : DWORD;
  pNameRVA : PDWord;
  pFunctionRVA : ^TFunctionRVA;
  pOrdinal: PWord;

begin
  // ...
  // 将文件读到Buff所指向的内存块

  // 定位到导出表
  p := Buff;
  if PosToSectionEntry(p, IMAGE_DIRECTORY_ENTRY_EXPORT, VABase, RawAddr)
then
  begin
    writeln('导出表起始地址: ', IntToHex(RawAddr, 8));

    inc(p, RawAddr);
    pBASE := DWORD(p);
    with PImageExportDirectory(p)^ do
    begin
      DWORD(pFunctionRVA) := pBASE + (DWORD(AddressOfFunctions) - VABase);
      DWORD(pNameRVA) := pBASE + (DWORD(AddressOfNames) - VABase);
      DWORD(pOrdinal) := pBASE + (DWORD(AddressOfNameOrdinals) - VABase);
```

```
    for i := 0 to NumberOfNames - 1 do
    begin
     writeln(format('Func: %.8x Ord: %.4x %s',
       [pFunctionRVA^[pOrdinal^],
       pOrdinal^ + Base,
       String(pChar(pBASE + (pNameRVA^ - VABase))^)]));
     inc(pNameRVA);
     inc(pOrdinal);
    end;
  end;
end;
// ...
```

上例中列出的 pFunctionRVA^值需要再经过运算才能得到有效的例程入口地址。除此之外，导出表结构并不复杂。但是，导出表的一些特例会导致分析过程变得更为复杂：

- 由于导出序号不需要连续，而 AddressOffunctions 所指向的又是一个下标连续的数组，因此，某些数组元素可能为空值。这表明对应序号的输出例程是无效(不存在)的。
- 由于 AddressOfNameOrdinals 所指向的数组仅表明 AddressOfNames 数组中对应例程的序号，它并不与 AddressOffunctions 中的元素一一对应。而有些例程仅用序号导出，因此，仅遍历 AddressOfNames 数组可能只能列举模块的部分输出。
- 通过调整 AddressOffunctions 中的值，可以使多个 Name 指向同一个序号的例程。因此并不能简单地通过 NumberOfFunctions-NumberOfNames 来得到仅用序号输出的例程数。如果要得到这样的值，需要遍历所有的三个数组并验证相互的对应关系。
- AddressOfFunctions 中存放导出例程入口的 RVA，但并不总是如此。按照约定，如果这个 RVA 值落在当前节中，则表明它是一个“输出转交(Export forwarding)”。例程的代码位于另一个模块文件中，RVA 值指向一个 PChar 类型的字符串，用来标示目标模块文件和导出例程的名字，其格式类似于：ntdll.EtwControlTraceA。

随书光盘中的 PECheck，用于检测这些特例，请阅读 CheckExportTable()例程及其相关的代码。

■ .EXE 和.BPL 中的导出表

.EXE 中也可以存在导出表，但这种使用方法并不常见。在 *Windows 2003* 的 *System32* 目录中的 4000 多个文件中，仅有 10 余个 *.EXE* 拥有导出表。其中 *ntoskrnl.exe* 的导出表有 1500 多个导出项。

使用关键字 exports，Delphi 中可以方便地在 *.EXE* 中导出例程和变量，这与在 *.DLL* 中导出例程和变量没有任何不同。

在.*EXE* 中导出的理由之一，是使得其他模块可以访问.*EXE* 模块(或者说 HOST 进程)中的例程与变量。这样的技巧通常应用在插件(Plugins)设计或者脚本引擎上。例如，第 7 章“内存管理的实现”中就使用这样的方法实现了 ShareMemRep。

在.BPL 中的导出表通常用来列举包中的类、类型和例程。Delphi 能够从导出表中分析出包中有哪些导出类，并且能够把在模块中对运行期库的访问变成导入表项。Delphi 在编译.BPL 时将自动生成类、方法和相关例程、类型的导出表，这样的机制是“包能导出类，而 DLL 不能导出类”的根本原因。

.BPL 中导出类时使用特殊格式的内部命名(Internal Naming)规则。以 *rtl70.bpl* 为例，其导出列表类同于：

```
C:\>tdump -ee -m c:\winnt\system32\rtl70.bpl
Turbo Dump  Version 5.0.16.12 Copyright (c) 1988, 2000 Inprise Corporation
                  Display of File RTL70.BPL

// ...
EXPORT ord:2545='@Classes@TComponent@'
EXPORT ord:2061='@Classes@TComponent@$bdtr$qqrp18Classes@TComponent'
EXPORT ord:2060='@Classes@TComponent@$bdtr$qqrv'
EXPORT ord:2059='@Classes@TComponent@BeforeDestruction$qqrv'
EXPORT ord:2023='@Classes@TComponent@ChangeName$qqrx17System@AnsiString'
EXPORT ord:2049='@Classes@TComponent@DestroyComponents$qqrv'
EXPORT ord:2048='@Classes@TComponent@Destroying$qqrv'
// ...
```

可见.BPL 导出时，可以有类、方法及单元信息。除此之外，按内部命名约定的规则，名称中还可以包括参数信息。命名格式为：

```
@[UnitName]@[MethodName]$[CallingConvention][ParameterType(s)]
```

其中，CallingConvention 规则调用约定的描述方式，如表 6-6 所示。

表 6-6　CallingConvention 规则调用约定的描述方式

标志字符串	调用约定	备　注
"qqr"	regsiter	缺省使用的调用约定
"Q"	pascal	单元名、过程名等将全部大写
"q"	cdecl	
"qqs"	stdcall	

表 6-7 是 ParameterType(s)用来描述参数类型列表。

表 6-7 ParameterType(s)用来描述参数类型列表

标志字符串	参数类型	备 注
"y"		没有参数
"2c"	ShortInt	
"s"	SmallInt	
"i"	Integer	
"l"	Longint	
"j"	Int64	
"uc"	Byte	无符号，单字节，字符类型
"us"	Word	无符号，双字节
"ui"	Longword	无符号，四字节，DWORD类型，Cardinal类型
"d"	Read	Decimal
"f"	Single	
"d"	Double	
"g"	Extended	
"c"	Char	
"pc"	PChar	
"o"	Boolean	
"6Real48"	Real48	
"11System@Comp"	Comp	
"17System@AnsiString"	String	
"15System@Currency"	Currency	

此外，ParameterType(s)中也包括参数声明的约定，如表 6-8。

① 类型标志字符串可以自定义，其格式为："[StringLength][ExprotString]"。表中所列出的这几种类型是符合"自定义类型标志字符串"的典型示例。可以使用同样的规则，描述所有非基本类型的例程参数——当然，它需要有一个被导出的名字。

表 6-8　ParameterType(s)中参数声明的约定

标志字符串	参数(x)声明	备　注
	x	没有标志字符串时，声明为值参数
"r"	var x	变量参数，传入变量地址
"x"	const x	
"r"	out x	传值参数，同变量参数一样，也是传入变量地址

这个标志串将附加在类型标志字符串之前。例如声明“var a: Integer”，则该参数的 ParameterType(s)将是“ri”。整个 ParameterType(s)字符串，是例程的全部入口参数声明按先后顺序串接而成的。

例如，在 *rtl70.bpl* 中有导出字符串(EXPORT ord:2044)：

```
'@Classes@TComponent@DefinePropertiesSqqrp14Classes@TFiler'
```

这表明在 *Classes.pas* 单元中，TComponent 的 DefineProperties 方法定义如下：

```
procedure DefineProperties(Var1: Classes@TFiler);regsiter;
```

其中“Classes@TFiler”是另一个导出项，可以在 *rtl70.bpl* 的导出表中查找到它的类型信息：

```
EXPORT ord:2578='@$xp$14Classes@TFiler'
```

导出名并不包含参数的名字，因此无法根据导出名来恢复确切的入口参数表。但已经能够知道如何来调用该例程了。因此，在执行 *tdump.exe* 时不加"-m"参数，以 C 语言规则导出的 *rtl70.bpl* 中，该方法描述如下(EXPORT ord:2044)：

```
'__fastcall Classes::TComponent::DefineProperties(Classes::TFiler *)'
```

此外，导出名没有指明例程是过程亦或函数的标志串。对于 Delphi 来讲，将函数作为过程调用并没有危险。但调用者有必要确知是否应该收集例程的返回值。如果为过程调用准备了返回变量，其结果是：该变量中将存放不确定的值。

6.3.4　数据节与代码节(DATA、BSS 和 CODE)

DATA 与 BSS 节都是数据节，不同之处仅在于存放的数据是否有初值，而并没有常量与变量之分。下面是一个 *.EXE* 的部分节表：

```
#   Name      VirtSize   RVA      PhysSize  Phys off  Flags
--  --------  ---------  -------- --------  --------  --------
02  DATA      00000584   00013000 00000600  00012400  C0000040 [IRW]
03  BSS       00000879   00014000 00000000  00012A00  C0000000 [RW]
```

DATA 的节属性是[IRW]，这表明它包含了一些数据的初值，因此操作系统仅需要直接映射 DATA 节到内存。而 BSS 节中的值是未初始化的(没有初值)，因此，操作系统映射之后，会将相应的内存映像充填 0 值。

由于实际 BSS 节是表明一个全 0 的数据块，所以它没有在物理文件中存在的必要。因此 BSS 节的 PhysSize 值为 0。但它在内存映像中却占有空间，其大小由 VirtSize 指定。

类型化常量和有初值的变量放在 DATA 节，而无初值的变量的 RVA 则指向 BSS 节。很明显，BSS 节不会导致物理文件大小的改变。因此，在源码中定义任意多个无初值的变量，是不会影响物理文件的大小的。

需要分配动态内存的类型化常量和有初值变量，其内存分配发生在 CODE 节的尾部。亦即是说，在 CODE 节中保存这些初始数据。但是，一旦这些常量或变量需要在运行期重新赋值，那么新的动态内存分配将发生在堆中，并且在 CODE 节中的原始值将被弃用。

需要在编译期就决定并在 CODE 进行动态内存分配的主要是指 “字符串的类型化常量”以及有初值的字符串变量。至于长度大于 4 字节的集合常量和字符串常量，尽管也是将初始数据放在 CODE 节，却未使用“动态内存分配”的规则。

准确地说，“字符串的类型化常量”与“有初值的字符串变量”在内存结构上并没有任何不同。而与普通的字符串变量比较起来，它们的内存占用的布局也是一致的，只是占用的内存位置不同。如图 6.6 所示。

应该留意在 CODE 节中的内存结构的引用计数与堆中是不同的。字符串常量的引用计数为 -1，而字符串变量的引用计数大于或等于 1。在内部例程_LStrAsg()中可以看到这个有趣的特征：

- 如果引用计数小于 0，则重新分配一个堆内存块并复制字符串。
- 如果引用计数等于 0，则释放字符串所占用的内存空间。

按照这样的处理方式，一个在 CODE 节中的字符串要么被复制，要么就被弃用，不可能被内存管理器释放。这避免了内存管理器对 CODE 节的误操作。

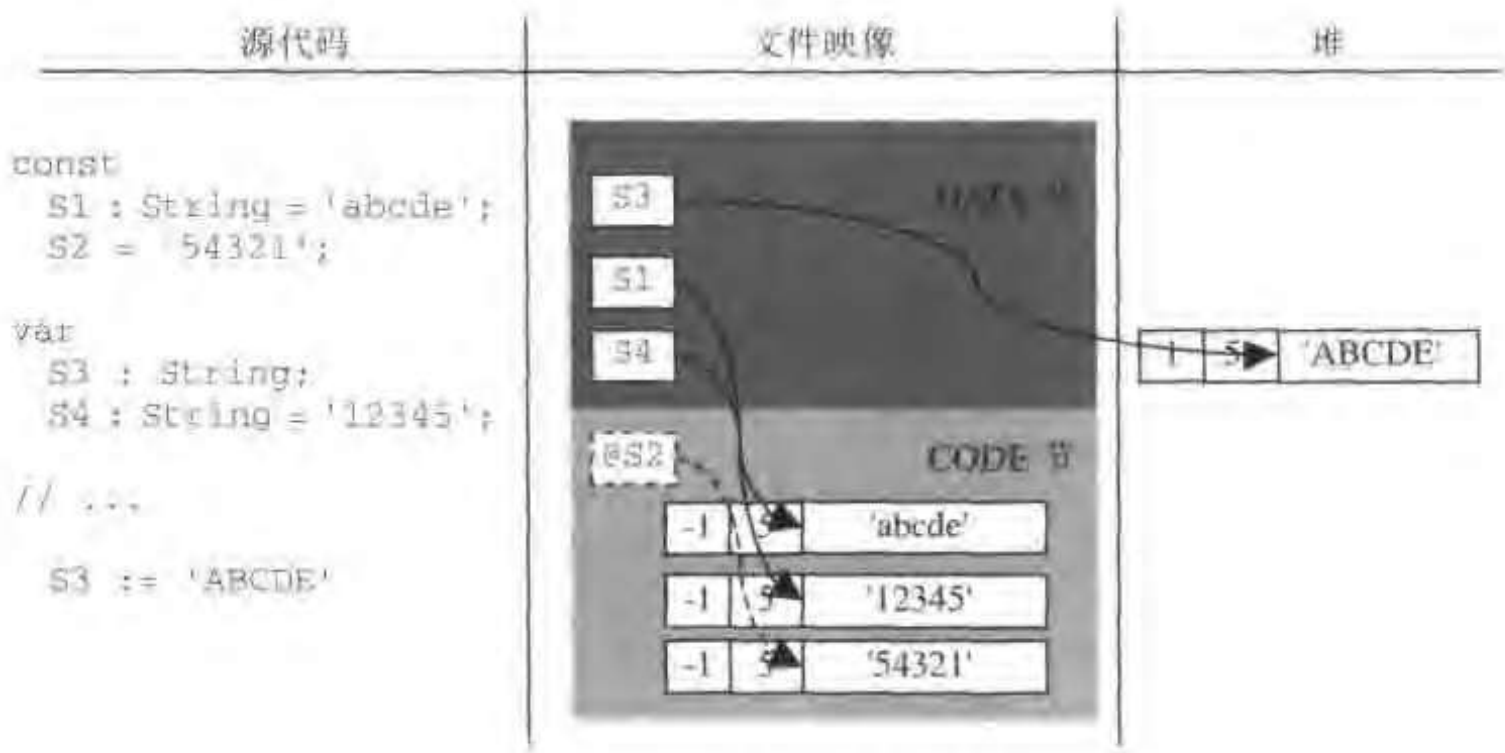


图 6.6　常量，变量在映像内存和堆中的布局

当然，这样看起来，字符串常量和有初值的字符串变量并不使用引用计数机制，它们被赋值时总会发生新的内存耗用。因此，一些情况下会导致系统效率下降。例如：

```
var
  Str : String = 'abcdef';
  StrArr : array [0..1000] of String;
  i : integer;

begin
  // 方法一
  for i := 0 to 1000 do
    StrArr[i] := Str;

  // 方法二
  StrArr[0] := Str;
  for i := 1 to 1000 do
    StrArr[i] := StrArr[0];
end.
```

表面上看起来，方法一和方法二的效果是一致的。但实际上，方法一将导致 1001 个新的字符串产生；方法二只会在 StrArr[0]中生成一个新的字符串，而其他 1000 个字符串都是指向它的引用。

明确地知道哪些数据在初始化时被放在 CODE 节中，意义非同寻常。因为某些技巧总是试图直接操作内存，而 CODE 节的属性是只读的，所以一些看起来有效的代码也可能导致内存存

取错误。例如：

```
var
  V_Str1 : String = 'abcdefgh';
  V_Str2 : String;
// ...
  V_Str2 := 'abcd';
  pChar(V_Str2)^ := 'a';  // 不出现错误
  pChar(V_Str1)^ := 'a';  // 出现错误
// ...
```

6.3.5 重定位节(.reloc)

只有在映像并没有按编译器预定设置的位置载入到内存时，才会使用到重定位节。

以.DLL 的入口代码为例，在编译期，编译器知道初始化表在映像中的位置，并以$00400000为基址假定的值，计算出一个 RVA 填入到物理文件中。因此，使用反编译工具可以看到的入口代码是这样的：

```
00407ECC Start:
00407ECC 55              push   ebp
00407ECD 8BEC            mov    ebp, esp
00407ECF 83C4C4          add    esp, -$3c
00407ED2 B88C7E4000      mov    eax, offset loc_00407e8c // 传入单元初始化表
00407ED7 E818C6FFFF      call   loc_00404534             // 调模块初始化例程
```

但这只是在物理文件中的情况，如果载入该 DLL 到进程中，则入口代码可能会是这样的：

```
Project1.dpr.27: begin
003B7EBC 55              push ebp
003B7EBD 8BEC            mov ebp,esp
003B7EBF 83C4C4          add esp,-$3c
003B7ED2 B88C7E3B00      mov eax,$003b7e8c
003B7ED7 E858C6FFFF      call @InitLib
```

在这种情况下，载入器需要将$00407FD2 位置上的指令作一个修改：修正$00407FD3 开始的四个字节($00407E8C)到新的值。这个过程就是重定位。为了完成这个过程，Delphi 的编译器在.reloc 节中加入一个重定位项，这个重定位项位于一个以_IMAGE_BASE_RELOCATION 结构开始的重定位目录中：

```
_IMAGE_BASE_RELOCATION = record
  VirtualAddress: DWORD;
  SizeOfBlock: DWORD;
```

```
    // WORD    TypeOffset[1];
  end;
```

可以有多个重定位目录，每一个的大小由 SizeOfBlock 的值决定，如果列举到一个 VirtualAddress = 0 的项时，表明重定位目录结束。

每一个重定位目录中，重定项数组位于结构的末尾(TypeOffset 域)。具体的项数为：(SizeOfBlock - SizeOf(_IMAGE_BASE_RELOCATION))/2，其元素类型为 WORD。这个 WORD 的高 4 位是重定位类型，其他 12 位则存放需要重定位的目标地址相对于 _IMAGE_BASE_RELOCATION.VirtualAddress 的偏移。

下面的代码简单地列出了所有重定位项的 RVA：

```
type
  _IMAGE_BASE_RELOCATION = record
    VirtualAddress: DWORD;
    SizeOfBlock: DWORD;
    // WORD    TypeOffset[1];
  end;
  PImageBaseRelocation = ^_IMAGE_BASE_RELOCATION;

var
  p: PByte;
  Buff: Pointer;
  i, tempSize : Integer;
  pBASE, VABase, RawAddr : DWORD;
  pRelocItem : PWORD;

begin
  // ...
  // 将文件读到Buff所指向的内存块

  // 定位到导出表
  p := Buff;
  if PosToSectionEntry(p, IMAGE_DIRECTORY_ENTRY_BASERELOC,
    VABase, RawAddr) then
  begin
    writeln('重定位表起始地址: ', IntToHex(RawAddr, 8));
    inc(p, RawAddr);
    while PImageBaseRelocation(p)^.VirtualAddress <> 0 do
      with PImageBaseRelocation(p)^ do
      begin
        tempSize := (SizeOfBlock - SizeOf(TImageBaseRelocation)) div
          SizeOf(WORD);
```

```
      DWORD(pRelocItem) := DWORD(p) + SizeOf(TImageBaseRelocation);
      for i := 0 to tempSize - 1 do
      begin
        writeln(format('RVA: %.8x, Type: %.4x',
          [VirtualAddress + pRelocItem^ AND $0FFF, pRelocItem^ SHR 12]));
        inc(pRelocItem);
      end;

      p := Pointer(pRelocItem);
      //DWORD(p) := DWORD(p) + SizeOfBlock;
    end;
end;
```

通常被重定位的数据位于 CODE 或者 DATA 节。重定位的算法也非常简单：

```
被重定位的原始数据 + (实际载入的内存映像基址 - PE可选头部中预设的ImageBase)
```

重定位是由于实际映像载入地址与预设的 ImageBase 不同而导致的。因此，绝大多数 Windows 内核级的.DLL 都使用不同的 ImageBase，以尽可能避免载入时的冲突。这样可以有效地提高.DLL 的装载速度。

对于 Delphi 编译的模块来说，.EXE 和.DLL 的 ImageBase 预设值都是$00400000，这使得.DLL 载入时都必然发生重定位。因此在效率要求较高的情况下(例如动态载入的插件)，应当对每个.DLL 设置不同的 ImageBase。

6.4 3.5K 的秘密

现在已经知道，Delphi 编译的模块最小为 3584 字节。那么，为什么大小会是这样呢？有没有可能编译出更小的模块来呢？

6.4.1 Delphi 的 PE 文件头部

对于 PE 文件来说，其头结构中的 DOS 头部和 PE 头部是不可少的。Delphi 标准的 DOS 头部为$100 字节。

接下来的 PE 头部包括识别标志、NT 文件头部、可选头部和数据目录。这三个部分也是定长的，具体的值为：

```
4 + IMAGE_SIZEOF_FILE_HEADER + IMAGE_SIZEOF_NT_OPTIONAL_HEADER
  = 4 + 20 + 224
```

```
= $F8
```

再接下来的是节表。因为节表之前的数据结构都是定长的，所以，节表的起始地址总是为：

```
DOS头部长度 + PE头部长度 = $100 + $F8 = $1E8
```

每个节表的项大小为40字节。而Delphi至少具有6个节，这意味着至少需要的节表长度为：

```
6 * IMAGE_SIZEOF_SECTION_HEADER = 6 * 40 = $F0
```

可以计算出Delphi的PE头部至少占用的空间为$1F8 + $F0 = $2E8字节。

6.4.2　Delphi的PE文件的节及其默认对齐

在可选头部的FileAlignment域中，Delphi的默认设置为$200。这表明编译器在生成PE头结构时，必须保证每一个节的物理地址起始于$200的倍数。

由于PE头部至少占用$2E8字节，那么，第一个节的起始地址最小值将会是$400。

在Delphi的6个默认节中，BSS节是不在物理文件中占用空间的。这表明有5个节必须在以$400为起始地址、每$200字节对齐的物理地址空间中存放。

因此，即使每个节都只有最少量的数据，那么最小化的物理文件长度必然是：

```
PE头对齐长度 + 节对齐长度 = $400 + $200 * 5 = $E00 = 3584
```

用TDump导出最小化的*.EXE*的节表，可以验证上述结论：

```
C:\>tdump MiniDExe.exe
#   Name     VirtSize   RVA      PhysSize  Phys off  Flags
--  -------- ---------- -------- --------- --------- --------
01  CODE     00000128   00001000 00000200  00000400  60000020 [CER]
02  DATA     00000004   00002000 00000200  00000600  C0000040 [IRW]
03  BSS      00000009   00003000 00000000  00000800  C0000000 [RW]
04  .idata   00000062   00004000 00000200  00000800  C0000040 [IRW]
05  .reloc   00000034   00005000 00000200  00000A00  50000040 [IRS]
06  .rsrc    00000200   00006000 00000200  00000C00  50000040 [IRS]
```

6.4.3　还可能更小吗

对于Delphi的编译器来说：

- DOS头部中的DOS执行程序是固定的。

- PE 头部是确定的、固定长度的数据结构。
- 节表项数与节的最小可能值是 6。
- 节的缺省对齐值为$200。

Delphi 的编译器必须依赖这四个基本规则来编译文件，因此没有任何可能使得编译的 PE 文件变得更小。如果必须要得到一个更小的文件，只能使用第三方工具，可能包括的手段有：

- 去掉 DOS 头部中的可执行代码。
- 调整缺省对齐值。
- 合并所有的节到同一节中。

即使如此，仍然至少需要一个大小为$200 的头部与一个$400 的节。这使得整个物理文件最小可能的字节数为$600 = 1.5 K。

当然，必要的前提是：如果有这样一个专用的优化工具的话。

6.4.4 3.5K 代码的内存映射

3.5K 的二进制数据最终会被载入内存。如果以$0040000 为基地址，那么，最小化的 Delphi 应用程序，会在内存中以怎样的映像存在呢？

借助 API：LoadLibraryEx()，可以简单地编写一段生成内存映像文件的代码：

```
program PELoader;
{$IMAGEBASE $72C00000}

uses Windows, Classes;

var
  Module : HModule;
  pDOS : PImageDosHeader absolute Module;

function GetImageSize : Integer;
begin
  if pDOS = nil then
    Result := 0
  else
    Result := PImageNtHeaders(DWORD(pDOS) +
      pDOS^._lfanew)^.OptionalHeader.SizeOfImage;
end;

begin
```

```
  if ParamCount < 1 then
    Exit;

  Module := LoadLibraryEx(pChar(ParamStr(1)), 0,
    DONT_RESOLVE_DLL_REFERENCES);
  with TMemoryStream.Create do
  begin
    Write(Pointer(Module)^, GetImageSize);
    Seek(0, soFromBeginning);
    SaveToFile(ParamStr(1) + '.IMG');
    Free;
  end;
end.
```

在随书光盘中的 **PELoader** 借助 **ImageHlp API**，使模块文件在载入之前能够被修正到指定的映像位置，从而可以得到模块在任意内存位置的映像文件。

PE 文件被载入进程空间之后，载入程序只需要查找到入口点，即可开始代码的执行了。下一节将描述所有情况下查找入口点的方法。对于操作系统来说，这是最后要做的事情，而对于 Delphi 的开发人员来讲，一切都才刚刚开始。

6.5　入口点

操作系统的载入例程从 PE 头中获悉文件对操作系统、机器硬件的需求，如果这些需求能被满足，则载入器将文件映射到内存中，调整好节的大小、位置，分配好初始堆、栈。

然后，载入例程从 PE 头中取得入口点(Entry Point)的信息，根据 *.DLL* 或者 *.EXE* 的不同约定开始调用入口点的代码。

找到入口点并按照约定调用入口代码，是操作系统在载入 PE 文件时所做的最后一件事情。自此之后，除非应用程序所在进程弹出异常，或者用户代码请求操作系统服务，其他所有的工作都将被代码节中的代码接管。

在 Delphi 的开发人员看来，这些就是程序的 *.pas* 要做的事了！

下面以 Delphi 在运行期载入的 *brcide.dll* 为例，使用一些工具来查看入口代码。操作系统查找入口点的方式与此类似。

6.5.1 磁盘文件上的入口代码(RAW Address)

使用 tdump 查看代码节的信息，下面是输出信息的一部分：

```
C:\>tdump E:\Delphi7\Bin\brcide.dll
#   Name      VirtSize   RVA      PhysSize  Phys off  Flags
--  --------  --------  --------  --------  --------  --------
01  .text     0000B000  00001000  0000AE00  00000600  60000020 [CER]
```

可以看到，代码节的 Phys off(物理偏移，也就是 RAW Address)为$600。这里，编译器将代码节的 RVA 设定为$1000，这是该节的相对虚地址的基点。

然后使用 tdump 查看入口地址：

```
C:\>tdump E:\Delphi7\Bin\brcide.dll | find /i "Entry RVA"
Entry RVA               000010CC
```

那么，可以知道入口点(Entry RVA)相对于代码节的基点的偏移为：

```
偏移 = Entry RVA - 代码节基点的RVA
     = $10CC - $1000
     = $CC
```

同样，在磁盘文件上，入口点相对于代码节的起始地址的偏移也应该是$CC，所以，它的起始地址为：

```
入口点RAW Address = 偏移 + 代码节基点的RAW Address
                  = $CC + $600
                  = 36CC
```

然后使用十六进制编辑器查看指定位置的数据（如图 6.7 所示）。



图 6.7 用十六进制编辑器查看指定位置的数据

记下查看到的物理数据，以便在后面的操作中印证该入口地址上的代码。

6.5.2　反编译器使用相对基地址定位的入口代码(RVA)

使用tdump查看编译器假定映像载入的内存地址(相对基地址)，如下：

```
C:\>tdump E:\Delphi7\Bin\brcide.dll | find /i "Image base"
Image base             00970000
```

由于入口代码的相对虚地址为**$10CC**，那么，在这个相对基地址之上的入口代码地址为：

```
RVA = Image Base + Entry RVA
    = $00970000 + $10CC
    = $009710CC
```

使用静态反编译工具(例如Win32Dasm)反编译 *brcide.dll*，在该地址上的代码为（图6.8）：

```
//********** Program Entry Point ********
:009710CC EB10                          jmp 009710DE
:009710CE 66623A                        bound di, word ptr [edx]
:009710D1 43                            inc ebx
:009710D2 2B2B                          sub ebp, dword ptr [ebx]
:009710D4 48                            dec eax
:009710D5 4F                            dec edi
```

图6.8

可以看到该地址上的十六进制数据正好与文件物理地址$6CC上的一致。

6.5.3　载入到内存之后(运行期)的入口代码(VA)

运行Delphi并使用工具listdlls在查Delphi32进程中查找到 *brcide.dll* 的载入地址：

```
C:\>listdlls delphi32 | find /i "version "
  Base             Size     Version         Path
  0x003c0000       0x1e000  7.00.0000.0453  E:\Delphi7\Bin\BRCIDE.DLL
```

可以发现，操作系统在Delphi32进程中装入 *brcide.dll* 时，并没有按(编译器)预期地那样载入到**$00970000**，而是载入到了**$003C0000**处。则运行期入口代码的虚地址：

```
虚地址VA = 运行期基地址 + Entry RVA
         = $003C0000 + $10CC
         = $003C10CC
```

使用动态调试工具(例如OllYDBG)载入Delphi32进程，可以查看到运行期 *brcide.dll* 的实际内存映像，在地址**$003C10CC**上的代码如图6.9所示。

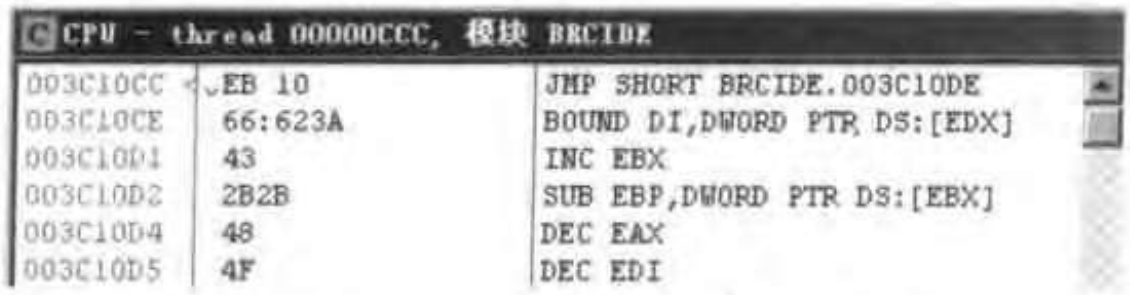


图 6.9 运行期 brcide.dll 的实际内存映像在地址$003C10CC 上的代码

可以看到：内存地址$003C10CC 上的代码，与反汇编工具以及十六进制编辑器中看到的代码，都是一致的。

第二部分 II

Delphi 内核深入剖析（II）

实现了内置数据结构、Win32 文件类型与 PE 文件结构，Delphi 已经可以开发出标准的 Win32 应用程序，并且具有一定的数据处理能力。但是这并不能很好地使用全部的操作系统资源。

Delphi 在内核一级进行封装，使得开发者能够方便地使用资源、线程、异常等操作系统资源，而无须过多地关心其细节。然而正是由于这样紧密的封装，使得开发者在试图利用 Delphi 跨过内核屏障，深入操作系统核心时会遇到重重阻碍。因此在某些情况下，不得不将这些封装从内核中剥离开来。

Nico Bendlin 在 MiniDExe 中大刀阔斧地将 System.pas 和 SysInit.pas 削减得只剩下真正意义上的“内核”，这使得我能够在下面的章节中一层层地包裹上 Delphi 的外装，回顾 Delphi 的上帝们创造这个神话的全部过程。

我不止一次地感叹：拥有全部的源码，的确是 Delphi 用户最引以为傲的事！

章节指引

接下来的章节讨论 System.pas 中对 OS 功能模块的封装，应该留意到 Delphi 如何使你能够操作 OS 中的全部资源。

第 7 章讲述内存管理器的数据结构和实现方法，同时讨论的内容还包括共享内存管理器和第三方的内存管理器。

第 8 章详细讨论 Delphi 的异常机制，准确地说是讨论 Delphi 如何封装操作系统的 SEH(结构化异常处理)机制。本章将深入分析面向对象的异常类的具体实现方法。

第 9 章是对多线程的深度剖析，涉及到操作系统、TLS 和 TThread 类的方方面面。

第 10 章讲述 Delphi 中内置的资源(Resouce)管理机制。但本章不包括对 VCL 窗体资源的讨论。

第 11 章通过三种应用的实现，讲述如何扩展 Delphi，使之输出跨编译器和跨平台的代码与模块。

第 7 章

Delphi 的内存管理器

最小化内核中并不包含内存管理器，这表明使用最小化内核开发应用程序，与使用汇编一样“原始”：需要使用标准的 Win32 API 来分配、使用和管理内存块。但是这通常会存在效率问题，也可能会导致内存漏洞。

如果代码中使用到需要动态内存的数据类型，那么这些数据类型的内部例程将访问 Delphi 的内存管理器。这些数据类型中就包括了 AnsiString。这意味着绝大多数的 Delphi 程序都需要内存管理器。

除了缺省的内存管理器外，Delphi 还实现了 *ShareMem.pas* 和相应的内存管理模块 *borlndmm.dll*，以同步 DLL 与 HOST 程序中的内存管理器。

与缺省的内存管理器一样，共享内存管理器(*borlndmm.dll*)并不是最有效率的。因此某些时候，选择第三方的内存管理器可能会得到更好的性能。

开发人员随时可以写一个内存管理器来嘲笑一下那个叫 *borlndmm.dll* 的东西。但在开始动手之前，请确认你已理解本章所述的全部细节，否则你可能不得不面对一次次的进程崩溃，并无可奈何地在 uses 关键字的后面加入 ShareMem。

知己知彼，才能无往不利。否则，笑到最后的还会是 Borland。

7.1 Delphi 的内存管理器实现框架

Delphi 在 *GetMem.inc* 中实现了自己的内存管理器。对于应用程序模块来说，除了变量、常量和系统内存的数据结构之外，其他任何时候的内存分配行为，只可能是如下三种情况之一：

- 通过操作系统 API 进行的进程内内存分配。
- 使用 Delphi 内存管理例程进行的堆分配。

- 通过汇编指令或者局部变量定义进行的栈分配[1]。

通过操作系统 API 进行的内存分配主要有虚地址内存分配和堆分配。前者是指使用 API：VirtualAlloc()进行的分配，后者是指用 API：HeapAlloc()进行的分配。

Delphi 内存管理例程中的所谓“堆分配”，与操作系统中的“堆”概念并不一致。Delphi 中指的是使用“堆”这种数据结构进行内存管理的方法，而操作系统中指的是一个堆内存区。[2]

在 *GetMem.inc* 中，Delphi 封装了操作系统 API，使得用户可以直接分配内存，而无须考虑内存具体在哪个虚地址空间、或者内存页中。

使用 Delphi 的开发人员不必了解任何有关操作系统内存分配的细节。对于这些开发者来说：任何一块(既定长度的)内存都是连续的，可以通过字节序遍历一个数据结构，或者在长度边界相容的情况下进行强制转换。

Delphi 通过图 7.1 所示的内存管理器框架对 API 实行封装：

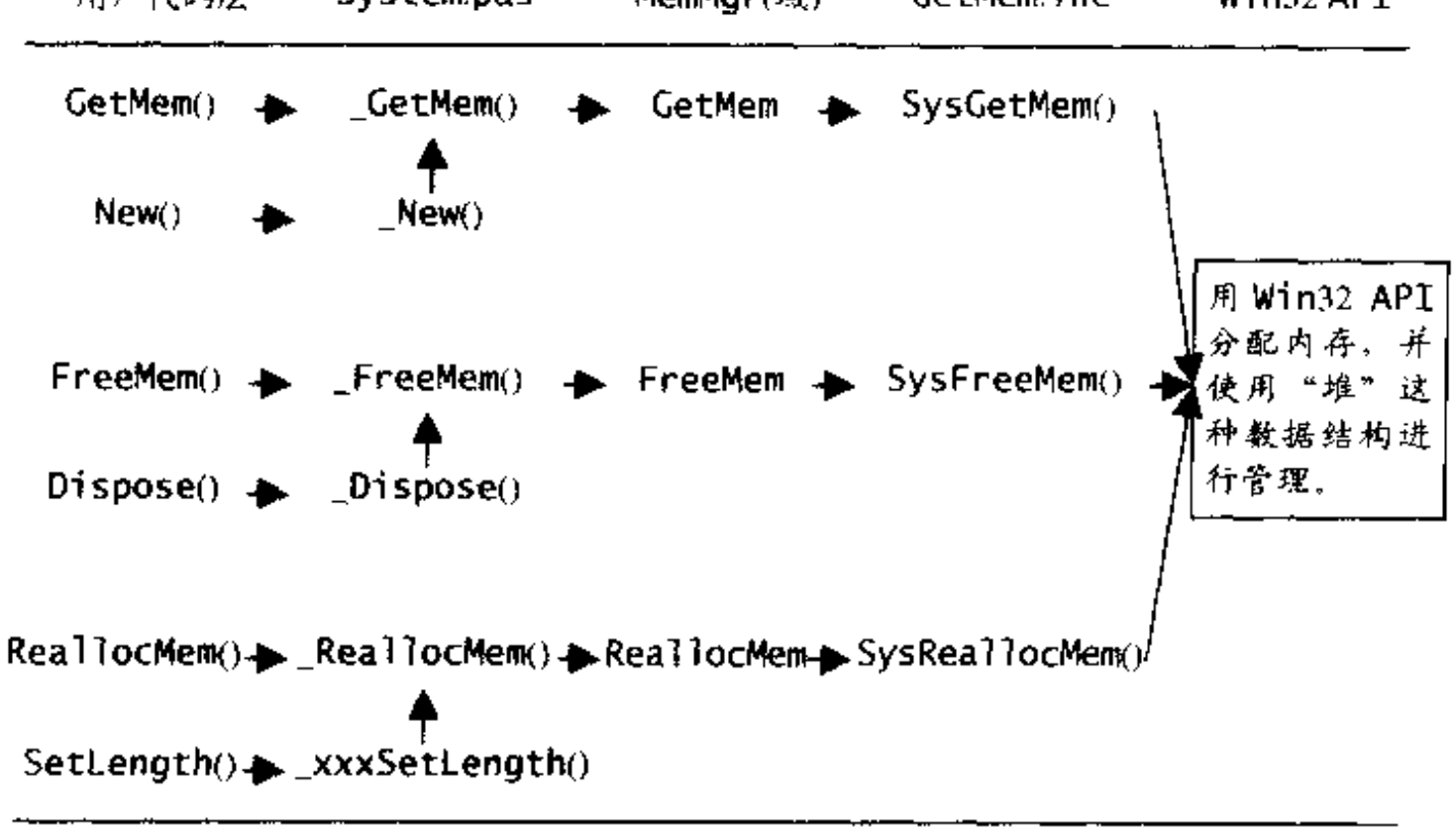


图 7.1　Delphi 内存管理器实现框架

在实现框架中，存在一些例外。例如_WStrSetLength()实现时，就不是通过调用_ReallocMem()来实现内存重分配的，而是通过_NewWideString()例程，调用

① 栈分配的相关内容请参见第 2 章和第 3 章。

② 除非特别指明，以下叙述的“在堆上分配内存”，都是指 Delphi 的“堆”内存管理方法，而不是操作系统的堆分配。

oleaut32.dll 中的例程 SysAllocStringLen()来实现。

7.2 内存页管理

内存页管理并非 Delphi 需要实现的管理机制，它是由操作系统实现的。

在 Win32 操作系统中，内存以“页”为单位进行分配。但内存页与内存虚地址空间是分开管理的。这表明，至少需要两步操作，才能得到一个可合法存取的内存块：

- 从整个虚地址空间中保留一个区间(Reserve)。
- 将一个内存块提交到虚地址空间上(Commit)。

这两步操作都使用同一个 API 来完成：

```
function VirtualAlloc(lpAddress: Pointer;
 dwSize, flAllocationType, flProtect: DWORD): Pointer; stdcall;
 external kernel name 'VirtualAlloc';
```

参数 flAllocationType 决定了进行哪一步的操作。其可能的取值为：

- MEM_RESERVE
 不分配任何内存空间，仅保留进程的虚地址空间的一个区间。

- MEM_COMMIT
 为指定的页范围分配物理存储[①]。具体是从物理内存或者是磁盘交换文件中分配，是由操作系统决定的。

另一个重要的参数是 lpAddress。其可能的取值为：

- nil
 在保留(Reserve)虚地址空间时，系统自行决定一个未分配的地址空间。

- 一个虚地址指针
 在保留(Reserve)虚地址空间时，以该值为参考，将地址舍入到下一个 64K 字节边界上。在提交(Commit)内存时，以该值为参考，将所分配的页舍入到下一个页边界。这意味

① 与大多数人所想象的不一样，Win32 API 中所说的物理存储(physical storage)，并不是机器内存插槽中的那些芯片(physical memory)。这里所说的物理存储，仅是相对于虚拟地址空间而言的。它实际上是指由物理内存、磁盘交换文件(*PageFile.sys*)所组成的系统可用的物理内存空间。

着实际分配的页，将是能够包含 lpAddress .. DWORD(lpAddress) + cbSize 范围的所有页。

依据操作系统的内存页管理机制，保留虚地址空间和提交内存都不一定正好是用户请求的地址和长度。因此，VirtualAlloc()例程将保证实际的分配大于或者等于用户所请求的虚地址空间或内存页。

正是因为这样的问题，所以在内存管理器实现中，将不得不随时修正堆块(Heap Block)的内存块边界。以充分利用 VirtualAlloc()提交的内存页上的未用空间。

7.3　堆

这里简要地说一下“堆”这种数据结构。

堆的主要构成：堆中包含许多大小不确定的块(Block)。

堆的主要操作：从堆中获取一个(至少为)指定大小的块；将一个空闲的块回收到堆；合并一些堆中的块以得到更大的块。

初始状态下，堆仅有一个块，即堆本身。经过一段时间地取用和回收以后，堆中将可能只剩下一些“切割”后残余的“碎片”，且这些碎片可能已经无法再合并。此时，如果一个新的请求大于任何一个碎片，那么就必须再申请一个新的、大的块放在堆中。

堆的使用，永远是一个“拆东墙补西墙”的过程。如果东墙拆得没有了，就去搬城砖。最终的结果是西墙不倒，或者城砖搬尽。

在最后一种情况下，Delphi 提示“Out of memory”。

7.4　MemoryManager 及相关例程

在 *System.pas* 中，MemoryManager 是一个有初值的记录变量。其定义如下：

```
var
  MemoryManager: TMemoryManager = (
    GetMem: SysGetMem;
    FreeMem: SysFreeMem;
    ReallocMem: SysReallocMem);
```

Delphi 源代码分析

该变量初值状态为：记录的三个域分别指向 SysGetMem()、SysFreeMem()和 SysReallocMem()。根据 Delphi 对有初值变量的处理原则[①]，在开始运行应用程序代码之前，内存管理器实际已经指向了有效的处理例程。这个动作发生在操作系统调用模块入口代码之前，当然也在单元初始化之前。

MemoryManager 是系统内部变量，不能够直接访问。*System.pas* 中提供三个与 MemoryManager 变量相关的例程：

```
procedure GetMemoryManager(var MemMgr: TMemoryManager);
procedure SetMemoryManager(const MemMgr: TMemoryManager);
function IsMemoryManagerSet: Boolean;
```

例程 GetMemoryManager()简单地复制 MemoryManager 记录的一个备份，而 SetMemoryManager()修改 GetMemoryManager()的出口变量 MemMgr 的域值，不会影响 MemoryManager 变量。

替换缺省内存管理器的惟一方法，就是调用例程 SetMemoryManager()。该例程只是简单地将用户的 MemMgr 复制到 MemoryManager 记录。这样的实现方法可以保证开发人员不能直接地操作 MemoryManager 变量。

Delphi 使用 GetMemoryManager()和 SetMemoryManager()这两个例程来实现 MemoryManager 的存取，仅是出于安全的考虑。

例程 IsMemoryManagerSet()用于检测是否有第三方的内存管理器设置：逐一比较 MemoryManager 的各个域，如果任意一个与初始不同，则表明已经修改了预设的内存管理器，返回 True。

据官方文档，如果需要替换内存管理器，则应当在调用 IsMemoryManagerSet()之前检测当前系统已分配内存的值。这是两个系统全局变量：

```
var
  AllocMemCount: Integer;    { 已分配内存的块数                }
  AllocMemSize: Integer;     { 已分配内存块的总长度(字节数)    }
```

下面这样的内存管理器替换代码是安全的：

```
var
  NewMemMgr, OldMemMgr : TMemoryManager;
```

① 参见 PE 结构中有关 DATA 节的描述。

```
// ...
// 设置NewMemMgr的域到有效的内存管理例程

GetMemoryManager(OldMemMgr);
if (AllocMemSize = 0) then
  SetMemoryManager(NewMemMgr);
```

大多数开发人员并不确知检测 AllocMemSize 的重要性。事实上，不检测 AllocMemSize 将导致极为严重的后果。

替换内存管理器的基本原则是：不能使用一个管理器例程去处理另一个(未知)管理器所分配的内存。如果 AllocMemSize<>0，则表明缺省的内存管理器已经启动，已经有内存被分配并存在于系统之中。此时替换到新内存管理器，则新内存管理器无法确知：后续代码是否会在切换回缺省内存管理器之前，尝试释放该内存块。因而这样的系统存在安全隐患。

一个简单的实例是：如果有字符串分别在新、旧两个内存管理器环境中被添加到 TStringList，那么在任何一个环境中调用 Free()，都会导致系统崩溃。

除非能够确知被替换的内存管理器的所有细节，标识出它所分配的内存并在需要的时候调用到 OldMemMgr 来处理，否则应当总是在检测 AllocMemSize 后替换内存管理器。

7.5　GetMem.inc 中的重要例程

MemoryManager 的三个域值例程实际位于 *GetMem.inc* 中。考虑到多线程环境，它们都使用了临界区变量 heapLock。因此在多线程环境中，Delphi 缺省的内存管理器的性能会相对较差一些。

GetMem.inc 首先针对“堆”这种数据结构，实现了堆块的操作例程。然后基于堆，使用三个层次结构来实现内存管理器，这包括：

- 虚地址空间(Address space)管理。
- 已提交的内存空间(Committed space)管理。
- 用户调用例程(actually calls)的实现。

7.5.1 堆块及其管理例程

■ 堆块：TBlockDesc(块描述)记录

每个堆块是某个双向链表的结点记录，数据类型定义为：

```
type
  PBlockDesc = ^TBlockDesc;
  TBlockDesc = packed record
    next: PBlockDesc;
    prev: PBlockDesc;
    addr: PChar;   // 并不表明这是一个PChar字符串，使用PChar类型仅为了方便地址运算
    size: Integer;
  end;
```

每 100 个 TBlockDesc 作为一个数据被存放在 TBlockDescBlock 中[①]，然后定义了基于 TBlockDescBlock 的单向链表 blockDescBlockList。

另一个结构是 blockDescFreeList。这是一个依赖 TBlockDesc.next 维护的单向链表，结点的其他域均未使用。更确切地说，这是一个空闲块描述结点的后入先出队列(LIFO)。

blockDescFreeList 总是指向表的头部。如果 blockDescFreeList=nil，则表明系统初始化或者系统所有申请的 n*100 个块已用尽，不再有空闲的 TBlockDesc。此时，例程 GetBlockDesc()将再次申请 TBlockDescBlock 结构，这意味着 100 个新的 BlockDesc 在进程的本地堆中产生：

```
// code from GetMem.inc, GetBlockDesc().

// ...
    bdb := LocalAlloc(LMEM_FIXED, sizeof(bdb^));
    if bdb = nil then begin
      result := nil;
      exit;
    end;
```

GetBlockDesc()接下来将这个 TBlockDescBlock 插入到 blockDescBlockList 链表的头部。然后，将初始化 *TBlockDescBlock.data*，使得该数组的初始情况为：

① 定义 TBlockDescBlock 的原因是有序表的访问会比链表访问更快，但这会导致 SizeOf(TBlockDesc) * 100 个字节的开销。

- data[0].next 等于 nil。
- data[1..99].next 指向上一个 BlockDesc，即 data[0..98]。

结束初始化循环的时候，blockDescFreeList 正好指向 data[99]，它将被 GetBlockDesc()作为返回值。这表明 Delphi 是以逆序访问 TBlockDescBlock.data 数组，取得空闲的 BlockDesc 结点的。

GetBlockDesc()在退出例程前，将返回结点的 next 域赋给 blockDescFreeList。显然，如果返回结点正好是 data[0]，则 blockDescFreeList = data[0].next，亦即是指向 nil。这使得下次调用 GetBlockDesc()时，将开始另一轮分配 100 个 BlockDesc 结点的循环。

```
function GetBlockDesc: PBlockDesc;
// ...
 if blockDescFreeList = nil then begin
   // 分配100个BlockDesc元素的数组，并初始化
   // blockDescFreeList指向最末一个BlockDesc，即data[99]
 end;
 bd := blockDescFreeList;
 blockDescFreeList := bd.next;
 result := bd;
end;
```

■ 非空闲的块描述结点

取得 BlockDesc 并不表明一个堆块的内存已分配。BlockDesc 仅用于记录堆中内存块或地址空间的相关信息。BlockDesc 也可以回收和合并，并总是以如下规则维护已用块描述链表：

- 若链表只有一个节点，则该节点的 prev 和 next 均指向自身。
- 若有 n 个节点，则：
 Node[n].prev->Node[n-1];
 Node[n].next->Node[n+1]。
 特例：
 Node[0].prev->Node[n];
 Node[n].next->Node[0]。

很明显，这构造了一个环形的双向链表。*GetMem.inc* 中用于维护这样的链表的例程有：

```
// 链表清空，使之初始化为只有一个(块描述)结点的链表
```

```
procedure MakeEmpty(bd: PBlockDesc);

// 为b所指向的一个内存块建立一个块描述，并将块描述插入到prev块描述结点之后
function AddBlockAfter(prev: PBlockDesc; const b: TBlock): Boolean;

// 从链表中取出指定块描述，然后将它插入到空闲块列表blockDescFreeList的头部
procedure DeleteBlock(bd: PBlockDesc);

// 从一个节点开始遍历链表，试图将表中与内存块b相邻的块合并到b中，
// 如果能够合并，则调用DeleteBlock()，将被合并的块描述符移到blockDescFreeList，
// 完成(尝试)合并操作后，将调用AddBlockAfter()将结果内存块b插入到prev块描述结点之后，
// 以下情况返回无效结果值result.addr = nil
//   1. 无法获取一个可用的块描述结点，因而无法将内存块b插入到prev块描述结点之后
function MergeBlockAfter(prev: PBlockDesc; const b: TBlock) : TBlock;

// 从一个节点开始遍历链表，检测每个块描述节点指定的地址空间是否能包含内存块b
// 如果内存块b被包含在某一节点(指定的范围)中，则从块描述节点中截去内存块b的地址空间
// 以下情况之一返回false:
//   1. 从链表中截去内存块b的地址空间后，剩余部分不能正确地插入到链表中
//   2. 无法在链表中找到符合条件的块描述结点
function RemoveBlock(bd: PBlockDesc; const b: TBlock): Boolean;
```

双向环形链表的特点是：无须计数链表的总长度，从任意结点(ANode)依序(next)访问结点，直到 Node.next 指向 ANode，则表明整个链表被遍历(以 prev 顺序访问的规则类同)。因此在例程 MergeBlockAfter()和 RemoveBlock()中，使用类似如下的代码，从入口参数 prev 结点开始遍历链表：

```
function MergeBlockAfter(prev: PBlockDesc; const b: TBlock) : TBlock;
var
  bd, bdNext: PBlockDesc;
begin
  bd := prev.next;
  repeat
    bdNext := bd.next;

    // ...
    // 如果内存块b与bd.addr是地址相邻的，则合并

    bd := bdNext;
  until bd = prev;  // 如果bd=prev，则表明使用next序完成遍历

// ...
```

用于保存块描述的双向环形链表在 *GetMem.inc* 中共有三个：

```
spaceRoot: TBlockDesc;          // 已分配的内存虚地址空间，其空间大小等于后两者的和
decommittedRoot: TBlockDesc;    // 未提交的物理内存大小(不能使用)
committedRoot: TBlockDesc;      // 已提交的物理内存大小(可用)
```

大多数情况下，*GetMem.inc* 使用上述变量的地址来作为链表的头部指针，并展开遍历。例如在 GetCommitted()例程中：

```
if MergeBlockAfter(@decommittedRoot, result).addr = nil then begin
  // ...
end;
```

任何一个 TBlockDesc 记录被创建之后就不会被释放，它总是存在于上述三个环形链表和 blockDescFreeList 单向链表四者之一中。

除了上述三个环形链表的首结点之外，所有的 TBlockDesc 记录都通过例程 GetBlockDesc()在本地堆中创建。所以，系统总的块描述符结点个数为 n*100+3 个。

TBlockDesc 结构及其相关的(对四个链表的)管理与内存分配本身是无关的。*GetMem.inc* 中仅用它们来管理内存地址，这包括“虚地址空间”和“提交的内存地址”两个方面。至于真正的内存分配，是在第三层“用户调用例程”中实现的。

7.5.2　虚地址空间(Address space)管理

GetMem.inc 中实现了如下五个虚地址空间管理例程：

```
// 调用VirtualAlloc()保留虚地址空间
function GetSpace(minSize: Integer): TBlock;

// 调用VirtualAlloc(), 从指定虚地址开始保留虚地址空间
function GetSpaceAt(addr: PChar; minSize: Integer): TBlock;

// 释放虚地址空间
function FreeSpace(addr: Pointer; maxSize: Integer): TBlock;

// 提交在(addr, minSize)地址区间中的所有块(所指定的页区域)
function Commit(addr: Pointer; minSize: Integer): TBlock;

// 取消提交在(addr, minSize)地址区间中的所有块(所指定的页区域)
function Decommit(addr: Pointer; maxSize: Integer): TBlock;
```

结果值 Result.addr<>0，表明调用 GetSpace()和 GetSpaceAt()成功，且返回的 Result 块已经通过调用 AddBlockAfter()例程添加到 spaceRoot 链表中。但 Result.Size 并不一定总等于 minSize 值。Delphi 根据如下的对齐规则来设定虚地址空间分

配的实际值：

- 如果 minSize 小于 cSpaceMin(1M Bytes)，则设为 1M Bytes；
- 如果 minSize 大于 1M，则在最小能容纳 minSize 字节的 cSpaceAlign(4K Bytes)边界上对齐。

FreeSpace()例程在 spaceRoot 链表中查找能够包含参数 "addr, maxSize" 所指定地址区间的块，并调用 VirtualFree()释放其保留的空间。虚地址空间释放后，即调用 DeleteBlock()将查找到的块从 spaceRoot 链表中清除。例程的返回值 Result 保存实际释放空间的起始地址和长度。

Commit()和 Decommit()都将进行边界对齐运算，提交和回收的值都是边界对齐后的地址和长度。返回值是实际操作的地址区间。

7.5.3 已提交的内存空间(Committed space)管理

对于已使用 Commit()提交的空间，将使用如下例程进行管理：

```
// 取至少为minSize长度的已提交块
function GetCommitted(minSize: Integer): TBlock;

// 取以addr为开始地址的，最小长度为minSize的已提交块
function GetCommittedAt(addr: PChar; minSize: Integer): TBlock;

// 释放已提交的地址空间
function FreeCommitted(addr: PChar; maxSize: Integer): TBlock;
```

以上三个例程实际上都是以 cCommitAlign 边界对齐值提交或释放。

GetCommitted 操作将首先遍历 decommittedRoot 链表，如果不能找到可供提交的空间，则调用 GetSpace()来保留一个新的地址空间。

由于调用 GetSpace()时，使用类似这样的代码：

```
result := GetSpace(minSize);
if MergeBlockAfter(@decommittedRoot, result).addr = nil then begin
  // 若地址空间无法登记到decommittedRoot，则调用FreeSpace()释放它。
  FreeSpace(result.addr, result.size);
  result.addr := nil;
  exit;
end
else
```

```
    // ok, 重新进行GetCommitted操作,
    // ...
```

在 GetSpace() 例程中，一个新的地址空间总是被加入到 spaceRoot 链表。因此在 MergeBlockAfter() 调用后，未提交的地址空间会在 decommittedRoot 和 spaceRoot 两个链表中存在重叠。而一个无法合并到 decommittedRoot 链表的空间将被立即释放——这通常是因为无法申请到新的 BlockDesc。

在 GetCommitted() 和 GetCommittedAt() 中使用这样的代码来完成提交：

```
function GetCommitted(minSize: Integer): TBlock;
var
  bd: PBlockDesc;
// ...
    result := Commit(bd.addr, minSize);
    if result.addr = nil then
      exit;
    Inc(bd.addr, result.size);
    Dec(bd.size, result.size);
    if bd.size = 0 then
      DeleteBlock(bd);
// ...
```

这使得提交以后，在 decommittedRoot 链表的块描述结点 bd 中，将剩下未被提交的高位地址空间。如果 bd.size = 0，表明相应的地址空间已经全部提交，则将其从链表中删除(使块描述空闲)。

在例程 FreeCommitted() 中，如果地址空间成功释放，则这个空间将被合并到 decommittedRoot 链表中。

7.5.4　用户调用例程(actually calls)的实现

如果地址空间被保留和提交，则意味着 Delphi 内核已经向操作系统成功地申请内存。接下来，*GetMem.inc* 中的大量代码用于实现 Delphi 内部的内存分配机制。

■ 内存边界对齐与实际分配长度

Delphi 向操作系统申请一个内存块的实际长度，总是大于用户代码中申请的值。除了内存边界对齐的因素之外，还因为 Delphi 需要使用一个 TFree 结构来保存内存块的大小和使用状态。因此如果申请的长度不能够容纳 TFree 结构，则将至少分配 SizeOf(TFree) 字节。这使得该内存块被释放时，能够作为一个 TFree 记录被缓存。

但是这并不是内存块的实际状态。如果一个内存块被申请并使用，那么该内存块将会包含一个 TUsed 结构。这个结构用来标识内存块的大小，以使内存块回收的时候，可以辨识是缓存还是调用 API 释放物理内存。这两个结构定义如下：

```
type
  PFree = ^TFree;
  TFree = packed record
    prev: PFree;
    next: PFree;
    size: Integer;
  end;
  PUsed = ^TUsed;
  TUsed = packed record
    sizeFlags: Integer;
  end;
```

对于任何一个已分配的内存块来说，一个 TUsed 结构总是被放在其头部。举例来说，一个 "GetLength(Str)" 为 10 的字符串，其内存占用的实际大小为：

```
StrSize    = 引用计数位 + 长度计数位 + 10Bytes + NULL结束符
           = SizeOf(Integer) * 2 + 10 + 1
           = 19
```

因此将调用 SysGetMem(19)来分配内存，在 SysGetMem 中加上 TUsed 长度：

```
AllocSize  = TUsed结构 + 19
           = 4 + 19
           = 23
```

但这仍然不是最终向操作系统申请的内存块长度。因为接下来还要进行边界对齐运算：

```
// 对齐值cAlign = 4
AlignSize  = (23 + (cAlign-1)) and not (cAlign-1)
           = (23 + 3) and not 3
           = 24
```

按照这样的内存分配规则，可以写出实际分配长度的计算函数，也可以从内存块指针的负偏移上得到这个长度值：

```
// define in GetMem.inc
type
  PUsed = ^TUsed;
  TUsed = packed record
    sizeFlags: Integer; // 必须通过运算才能得到实际长度
  end;
```

```
const
  cAlign         = 4;
  cThisUsedFlag = 2;
  cFillerFlag    = Integer($80000000);
  cPrevFreeFlag = 1;
  cFlags         = cThisUsedFlag or cPrevFreeFlag or cFillerFlag;

// 取指针所指向内存块的对齐长度
function GetAlignSize(const p : pointer) : integer; overload;
var
  u : PUsed;
begin
  u := p;
  dec(u);
  result := u.sizeFlags and not cFlags;
end;

// 取字符串的内存占用的对齐长度
function GetAlignSize(const Str : string) : integer; overload;
var
  p : PInteger;
begin
  p := Pointer(Str);
  dec(p, 2);
  Result := GetAlignSize(Pointer(p));
end;

// 计算对齐长度
function CalcAlignSize(Size : Integer) : Integer;
begin
  Result := (Size + SizeOf(TUsed) + (cAlign-1)) and not (cAlign-1);
  if Result < 12 then  // SizeOf(TFree) = 12
   Result := 12;
end;

// ...
begin
  GetMem(p, 10);
  Writeln('直接获取: ', GetAlignSize(p));
  Writeln('  计算值: ', CalcAlignSize(10));

  Str := '0123456789';
  Writeln('直接获取: ', GetAlignSize(Str));
  Writeln('  计算值: ', CalcAlignSize(SizeOf(Integer)*2 + Length(Str) + 1));
```

Delphi 源代码分析

```
// ...
```

在上例中，TUsed.sizeFlags 按照如图 7.2 所示的格式存储，从该域提取内存长度值时，依赖于一些标志位的运算。

GetMem.inc 中通常使用 BlockSize 来表明对齐长度(AlignSize)。基于 Delphi 的内存实现机制，将无法确切地知道对齐前的长度值。

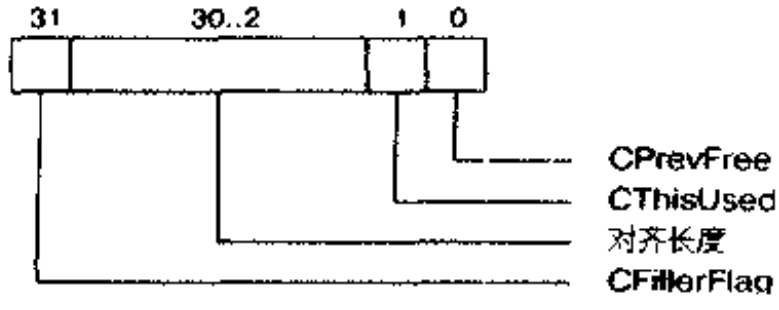


图 7.2 TUsed.sizeFlags 的存储格式

■ 内存块运行期状态

如前所述，用户每次申请的内存块会在负偏移处有一个 TUsed 结构，这使得整个内存块大小(BlockSize)为 "SizeOf(TUsed) + AllocSize" 的对齐值。

一个被释放的内存块除了头部被修改成 TFree 结构之外，它的尾部也会被修改成一个 TFree 记录。这使得不同状态的内存块在运行期结构如图 7.3 所示。

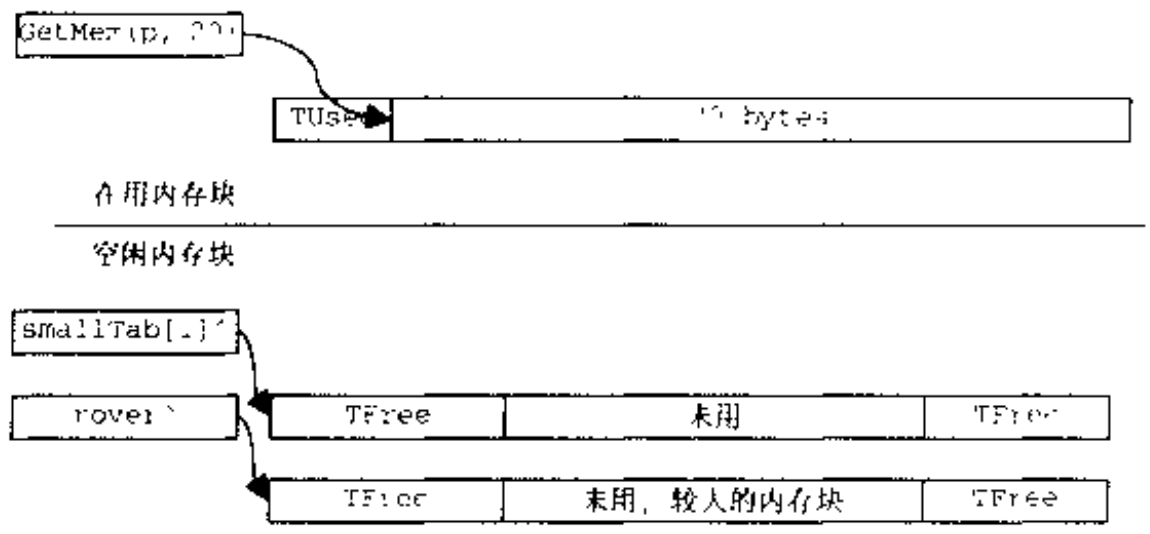


图 7.3 不同状态下的内存块结构

在目前仅见的一些关于内存管理器的资料中[1]，空闲内存块尾部被描述成 TUsed 结构，这

[1] 截止成书之日(2004.06.23)，我能查到的惟一一份资料是俄文的，它发表在《RSDN Magazine #2》(2003.02.21)。我当然不会俄文，但是依据它所提供的图例来看，它对空闲内存块尾部结构的描述是有误的。

并不正确。

GetMem.inc 中所理解的 TFree.size 与 TUsed.sizeFlags 是不同的。前者是一个真实的长度值，后者却包含了三个标志位，需要使用 sizeFlags and not cFlags 运算才能得到有效的长度值。但在内存块末尾的数据结构的长度值并不是 sizeFlags，而是确切的长度值，因此它是 TFree 结构。

但是这样将导致一个更难以理解的问题：如果尾部是 TFree 结构，那么要容纳首尾两个 TFree，则块长度至少需要 12*2 个字节，而实际最小的块长度 CalcAlignSize(1)仅为 12 个字节。

对于长度为 12 字节的块，它首、尾的两个 TFree 结构是重叠的，因此无须讨论。而接下来的对齐长度为 16 字节，这便是尾部结构被误解为 TUsed 的原因了。其实，尾部 TFree 结构的 prev 和 next 两个值是根本不用的，所以在对齐长度为 16、20 字节的情况下，尾部与首部的 TFree 的部分域重叠，却也不会冲突。——这里，对结构的不同理解都是为了方便地址运算。

■ 空闲内存块的合并

在内存块的尾部添加一个 TFree 的目的，是使得 Delphi 可以通过尾部的 TFree.size 来查找到头部的 TFree。这样，在释放一个内存块时，GetMem.inc 中的 SysFreeMem()就可以根据下面的规则来实现向前合并：

- 如果当前内存块的 TUsed.sizeFlags 的 cPrevFreeFlag 置位，则低地址端相邻的内存块是空闲的；
- 从(低地址端)相邻的内存块尾部的 TFree 结构中取得 size 值；
- 查找到(低地址端)相邻内存块头部的 TFree 结构；
- 合并两个内存块，并试图释放合并后的内存块。

很显然，由于当前内存块的 cPrevFreeFlag 可以表明相邻的前一个内存块的状态，所以总是能安全地访问前一个内存块。但是除了向前合并之外，在释放内存块时，Delphi 还试图向后合并。那么 Delphi 又如何知道后一个内存块的状态呢？

后一个内存块头部的四个字节，只可能存在两种情况：

- 一个空闲内存块头部的 TFree.prev 域；
- 一个在用内存块头部的 TUsed.sizeFlags 域。

由于 TFree.prev 指向的总是另一个内存块的地址值(或 nil)，而地址值总是以 4 Bytes(cAlign)为边界对齐的。因此，如果是 TFree.prev，则其低二位为 0。对于

TUsed.sizeFlags 域来说，低二位分别为 cPrevFreeFlag 和 cThisUsedFlag 标志，那么，如果内存块是“在用的”，则 TUsed.sizeFlags 的 cThisUsedFlag 标志位为 1。因此，仅需要检测后一个内存块头部四字节的 cThisUsedFlag 标志域，就可以识别内存块的状态。

所以在 SysFreeMem()例程中，用这样的代码来检测和合并后一个内存块：

```
// code from GetMem.inc, SysFreeMem()

  // 相邻的后一个内存块
  n := PUsed(PChar(u) + size);

  // ...
  if (n.sizeFlags and cThisUsedFlag) <> 0 then begin
    // ...
    // 后一个内存块在用，则设置它的"前一个内存块被释放"的标志
    n.sizeFlags := n.sizeFlags or cPrevFreeFlag
  end else begin
    // 后一个内存块空闲，则向后合并
    // ...
  end;
```

■ 内存释放(FreeMem)

与内存释放相关的全部例程如下：

```
// FreeMem()例程
function SysFreeMem(p: Pointer): Integer;

// 用于在内部调用SysFreeMem()例程
procedure InternalFreeMem(a: PChar);

// 将a指向的，size大小的内存块插入到smallTab数组，avail链表中，或调用
// DecommitFree()释放
procedure InsertFree(a: Pointer; size: Integer);

// 填前面的Gap
procedure FillBeforeGap(a: PChar; size: Integer);

// 填后面的Gap
procedure FillAfterGap(a: PChar; size: Integer);

// 从smallTab或avail链表中去掉一个块
procedure DeleteFree(f: PFree);
```

```
// 例程FindCommitted()用于在committedRoot查找包含addr的已提交的内存空间，返回
// 该内存空间的块描述。仅DecommitFree使用本例程。
function FindCommitted(addr: PChar): PBlockDesc;

// 释放较大内存块的物理存储
function DecommitFree(a: PChar; size: Integer): Boolean;

// 释放当前分配的内存块(参见下一小节)
procedure FreeCurAlloc;
```

如果用户调用 SysFreeMem()例程释放一个内存块，那么，这个内存块整体(包括可以合并的前、后空闲内存块)将作为一个 TFree 记录：

- 如果块长度小于 4K，则插入到 smallTab 数组中的某一个环形链表中；
- 如果块长度小于 15K(cDecommitMin)，则插入到 avail 指向的链表中；
- 否则调用 DecommitFree()将其释放。

SysFreeMem()中将调用例程 InsertFree()，来实现上述的操作。

收集块长度小于空闲内存的 smallTab 数组，是一个非常复杂的数据结构：

```
const
  cAlign          = 4;
  cSmallSize      = 4*1024;
type
  TSmallTab       = array [sizeof(TFree) div cAlign .. cSmallSize div cAlign]
    of PFree;
VAR
  smallTab        : ^TSmallTab;
```

smallTab 数组的每一个元素均指向一个 TFree 结点类型的双向环形链表，数组的下标为[3..1024]。*GetMem.inc* 用这样的一个公式来计算指定长度的内存块在 smallTab 数组中的下标值：

```
AlignSize := CalcAlignSize(size);
i := AlignSize div cAlign;              // i是访问smallTab时使用的下标
```

虽然 smallTab 是使用 API：LocalAlloc()在进程的堆上分配的，但在它的每个元素所指向的环形链表中，却是使用 VirtualAlloc()分配的内存块。这些内存块也就是用户使用 GetMem()所得到的内存块。

对于块长度(BlockSize)小于 15K 的空闲内存块，Delphi 总是试图将它作为一个 TFree 结构插入到 avail 链表中。为了快速访问 avail 链表中的内存块，*GetMem.inc* 中定义了一个指针 rover。

一个空闲内存块被释放时，它总是被插入到当前 rover 指针之前，然后重置 rover 指针到新释放的内存块。这保证了 rover 指针总是指向最近释放到 avail 链表中的空闲内存块[1]，而距离 rover 指针的 next 序越远，则表明该内存块越早被释放。这样，在遍历 avail 链表时，只要以 rover 指针开始访问，就可以有最大的几率利用操作系统的内存缓存机制。

对于长度大于或等于 15K 的空闲内存块，InsertFree() 例程将试图调用 DecommitFree()，在 DecommitFree()中则调用 FreeCommitted()来实现释放物理存储和地址空间的功能。

接下来的问题将因为内存分页和对齐机制变得越来越复杂。

首先，FreeCommitted()在释放已提交空间时，需要进行边界对齐运算。因此，入口时的 addr 和 maxSize 就可能不是真实释放的值。该例程返回 TBlock 类型的值来存放实际释放的 addr 和 size。如果不能成功释放，返回 Result.addr=nil，这时，InsertFree()会将这个块插入到 avail 链表中。

因此，一个(大的)内存块在释放时可能发生的情况如图 7.4 所示。

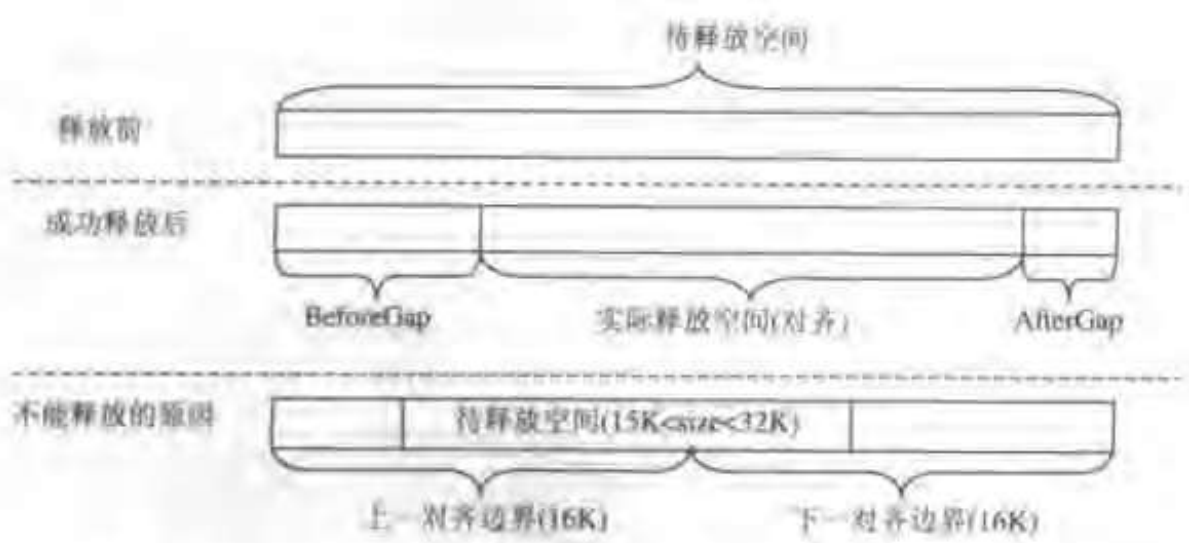


图 7.4 调用 FreeCommitted()例程的实际效果

内存管理器必须有一种机制来识别和处理这些空隙(Gap)。这就是 *GetMem.inc* 中下面两个例程的设计目的：

```
// 填前面的Gap
procedure FillBeforeGap(at PChar; size: Integer);
```

[1] 基于操作系统的内存管理机制，重新访问最近操作过的内存块，总是效率最高。例如一个 *pagefile.sys* 中的内存页可能被缓存在物理内存中，如果重新访问它，则不必再从磁盘中读取。此外，CPU 会进行高速缓存地址转换(虚地址->页面地址)，重复访问同一地址就可以不须要重新转换。

```
// 填后面的Gap
procedure FillAfterGap(a: PChar; size: Integer);
```

对于 BeforeGap：

- 如果 Gap 大小在 4~15 字节，则处理成一个 Gap 记录；
- 如果 Gap 大小超过 15 字节，则处理成一个空闲内存块(TFree)，并在尾部补充一个 Gap 记录。然后调用 InsertFree()，插入到 smallTab 或者 avail 链表中。

对于 AfterGap：

- 如果 Gap 大小在 4~11 字节，则处理成一个 Gap 记录，且后一个内存块的 TUsed.sizeFlags 的 cPrevFreeFlag 位将被清除[①]；
- 如果 Gap 大小超过 11 字节，则处理成一个在用内存块(TUsed)，然后调用 InternalFreeMem()例程重新释放。

因此，在空隙处理之后，原来的待释放空间将变成如图 7.5 所示的结构。

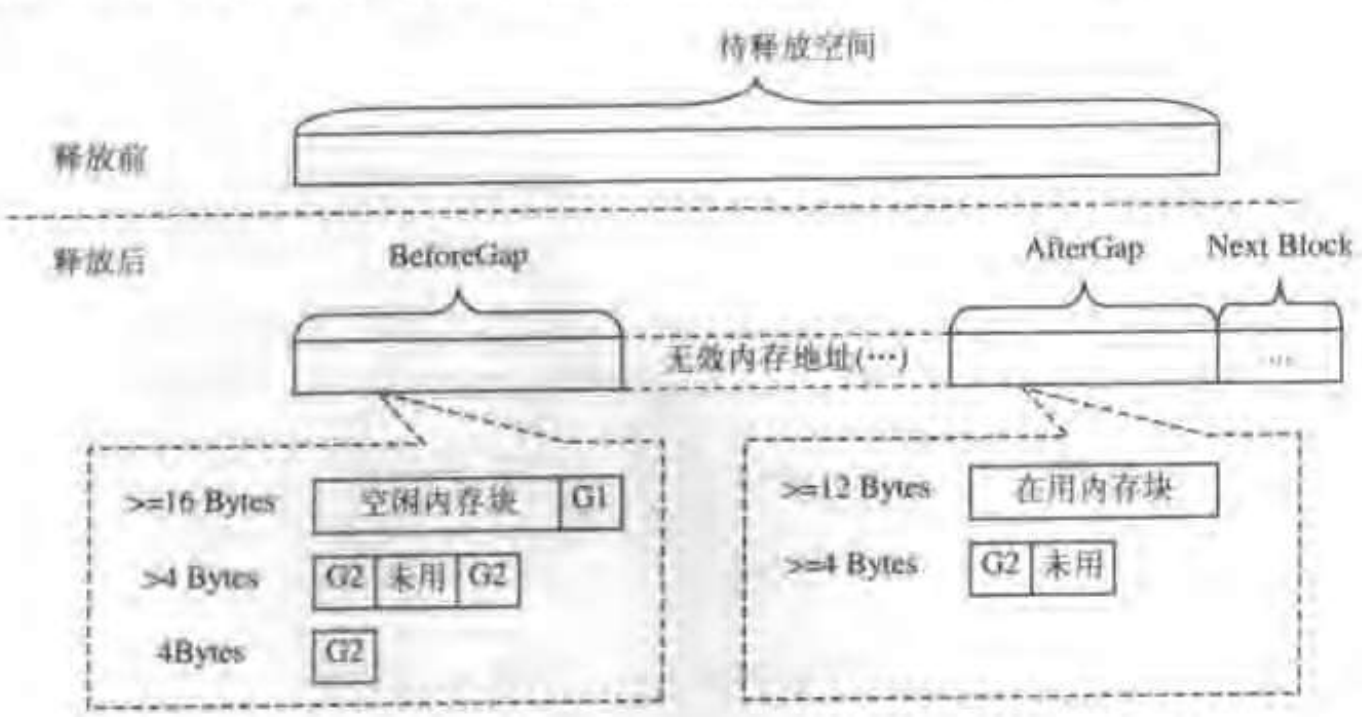


图 7.5　空隙处理后的效果

Gap 记录实际是 TUsed 结构，因此 Gap 记录也保存着 Gap 的长度值。图 7.5 中，G1 和 G2 是指不同的 Gap 类型，这主要表现在 sizeFlags 域的一些标志上：

- G1：cPrevFreeFlag 置位。这用于指明 G1 前有一个空闲内存块。cThisUsedFlag、

① 很明显，这两处存在了不可再分配的碎片，只有出现新的空闲块，并能够将它们重新合并，才有可能重新被分配或者释放地址空间。

cFillerFlag 也同时置位。

- G2：仅 cThisUsedFlag、cFillerFlag 标志置位。是用于填充的普通 Gap。

在 FillAfterGap()例程中，如果 GapSize >= 12，那么使用如下代码填写在用内存块头部的 TUsed 结构：

```
PUsed(a).sizeFlags := size or cThisUsedFlag;
```

这样，cPrevFreeFlag 位缺省值就是 0，这使得接下来的 InternalFreeMem()操作中，不会试图向前合并。同样的原因，必须考虑：如果 Gap 足够小，则将来需要释放 AfterGap 之后的 Next Block 时，也不能试图向前合并。因此 FillAfterGap()使用如下的代码来强制填 Next Block 的 sizeFlags 域：

```
// 下一个内存块的cPrevFreeFlag取消置位
Inc(a,size);
PUsed(a).sizeFlags := PUsed(a).sizeFlags and not cPrevFreeFlag;
```

■ 内存分配(GetMem)

开发人员总有一些固有的操作内存块的习惯。例如下面的这样一段代码：

```
procedure TestMemAlloc;
var
  Str : string;
begin
  Str := '0123456789';
  // ...
end;
```

在这个例程中，需要在堆上分配块长度为 24 字节的内存。但是例程结束时，如果字符串 Str 没有被引用，那么它所占的内存就将被立即释放。由此可见，在应用程序中，“释放一个刚分配的内存块”的概率是很高的[①]。然而如果每次释放都将它放到 smallTab 或者 avail 链表，然后需要时再查找，则效率将会变得很低。

因此，curAlloc 指针被用来保存当前分配的一个内存块：

```
VAR
  remBytes      : Integer;      // curAlloc指向内存块的大小
```

① 这样的情况经常发生在循环调用例程或者调用递归过程中。

```
  curAlloc      : PChar;          // 当前(上一次)分配的内存块指针，用于快速访问

// 释放当前(上一次)分配的内存块。有两种情况会导致CurAlloc的释放：
//   1，当前请求的内存块大于CurAlloc，需要重新提交内存时；
//   2，释放内存时，合并后的CurAlloc大于cDecommitMin。
// 释放操作实际上是将块插入到smallTab，avail或者调用DecommitFree()释放。
procedure FreeCurAlloc;
```

这样，有内存块释放时，先试图合并到这个内存块上，直到它大于 15K，才调用 FreeCurAlloc()例程释放其物理存储。如果有新的内存块需要分配(GetMem)，(在访问 avail 链表之前)也总是尽可能先在 curAlloc 中进行。这使得总长度小于 15K 的连续的内存分配、释放动作变得非常高效。

用于内存分配的主要例程包括：

```
// GetMem()例程
function SysGetMem(size: Integer): Pointer;

// 从smallTab，avail链表中分配指定长度的内存，或重新向OS申请一个内存块
function TryHarder(size: Integer): Pointer;

// 在smallTab中查找至少为size字节的块
function SearchSmallBlocks(size: Integer): PFree;

// 提交新的物理存储，新提交的块总是出现在curAlloc指针上
function NewCommit(minSize: Integer): Boolean;

// 合并新提交的块到committedHead，并将块首地址赋给curAlloc指针
function MergeCommit(b: TBlock): Boolean;
```

GetMem.inc 中，内存分配总是尽可能遵循这样的一些原则：

- 避免从大内存块中切分内存，以防止碎片产生；
- 避免访问 avail 链表，因为这是个无序的链表，访问效率较低；
- 避免调用 NewCommit()重新提交物理存储，因为这将导致内存开销；
- 在 NewCommit()中，避免调用 GetSpace()重新划分空间，因为这除了导致内存开销之外，还将导致新一个页的地址空间占用。

因此，SysGetMem()例程：

- 总是先访问 smallTab，直接查找正好满足长度需求的小内存块；如果不成功，则，
- 访问 curAlloc 指针，直接从头部截取指定长度的内存块；如果不成功，则，
- 调用 TryHarder()，访问 avail 链表，试图查找满足长度需求的内存块，切分；

如果不成功，则，

- 再次访问 smallTab，查找大于需求长度的内存块，切分；如果不成功，则，
- 调用 NewCommit()提交一个新的块，如果出错则退出 SysGetMem()；否则查找新块是否分配在 curAlloc 指针中；如果不成功，则，
- 重复上述的分配规则。

可以想象，avail 中总是比较大的块，所以通常情况下，avail 中找不到的块，在 smallTab 中也不可能找到。那么 TryHarder()例程中，为什么在访问 avail 链表之后，还要再调用 SearchSmallBlocks()来访问 smallTab 呢？这是因为存在一种特殊情况：avail 表只有一个初始结点，该结点 avail.size = 0。这时就有必要访问 smallTab 了。

在 TryHarder()中，访问 avail 链表时并没有采用这样的结构：

```
if size < cDecommitMin then
begin
  // 遍历avail链表
end;
```

这是因为在调用 InsertFree()时会进行边界对齐运算，因此会将长度不小于 cDecommitMin 的、但调用 DecommitFree()又无法释放的块插入到 avail 链表，使得 avail 链表中存在比 cDecommitMin 值更大的块。

但是，依据对齐的原理，一个块如果不小于 2*cCommitAlign，则一定会被释放(部分或全部)。因此，在 avail 中查找这样的块是没有意义的，没有对此加以排除，是 TryHarder()例程在效率上的一个疏忽。

如果在 avail 和 smallTab 中都得不到一个可用的内存块，则 TryHarder()将调用 NewCommit()来分配新的内存块，新的内存块总是出现在 curAlloc 指针上，原 curAlloc 指针所指向的内存块，将通过调用 FreeCurAlloc()例程插入到 samllTab、avail 或者调用 DecommitFree()释放。

任何一个通过 NewCommit()得到的内存块，都将先试图与 committedRoot 链表中已提交的块合并，这是通过 MergeCommit()例程来实现的。这个合并操作与上一小节中提到的 DecommitFree()例程的机制息息相关。一个极端的例子是：应用程序先释放了一些物理存储，调用 DecommitFree()时正好释放了一个页；接下来，应用程序又需要申请一个新的页空间(如果新内存块的需求足够大)，则上一次释放的页正好被重新申请到。这时，堆中出现的情况如图 7.6 所示。

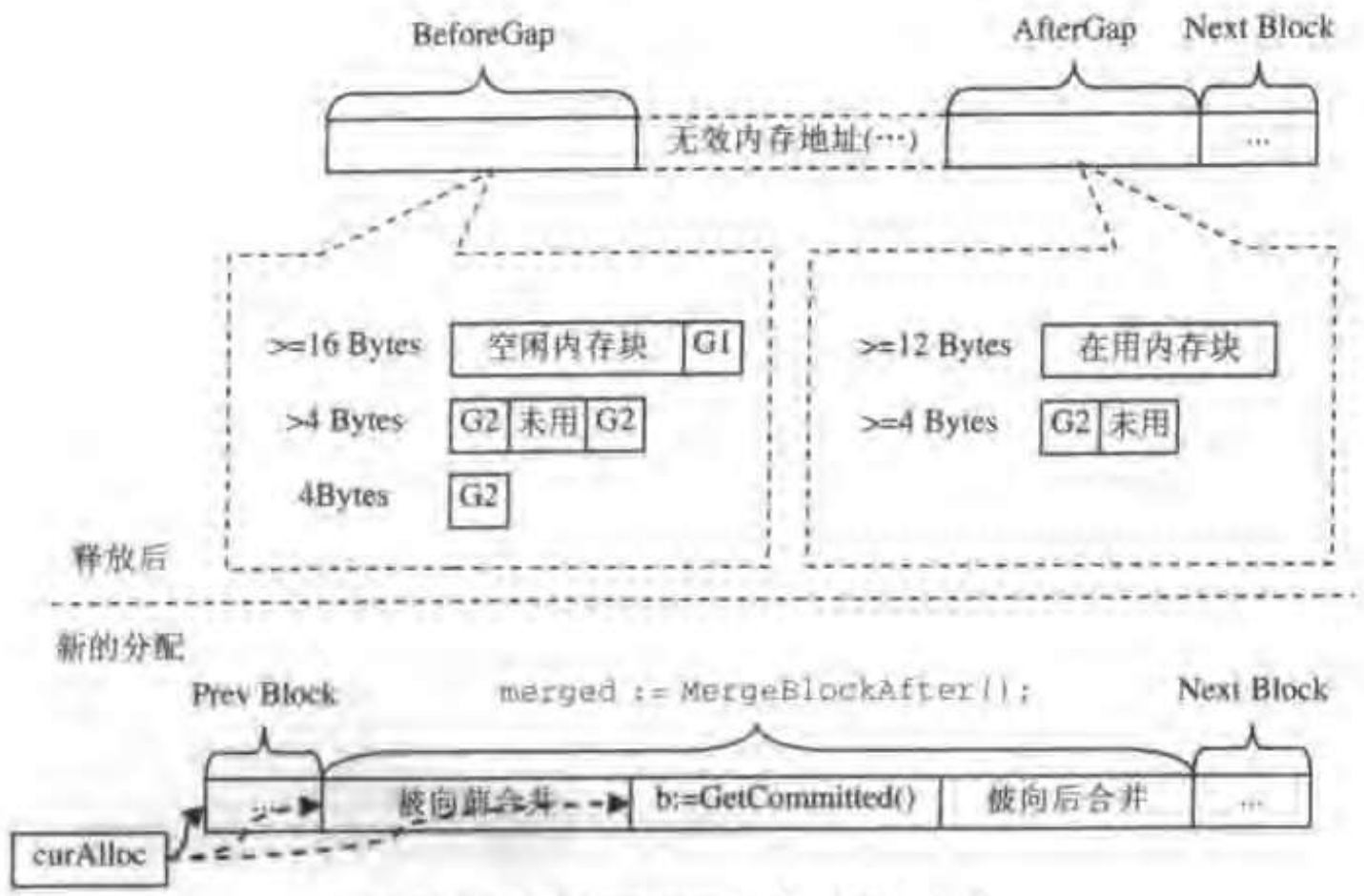


图 7.6　调用 NewCommit()的实际效果

在这个图例中，AfterGap 中的“在用内存块”最终会被调用 InternalFreeMem()释放，这使得它也将成为一个空闲内存块并被合并到 merged。

MergeCommit()例程用来调用 MergeBlockAfter()并检测合并的结果：

- merged.addr < b.addr，向前合并；
- merged.addr + merged.size > b.addr + b.size，向后合并；
- merged.addr + merged.size = b.addr + b.size，合并后块 b 的末端正好在对齐边界上(需要在末端则填一个 Gap)。

这里发生的向前、向后合并是调用 MergeBlockAfter()时，通过遍历 committedRoot 形成的。接下来调用的两个例程用来检测合并的有效性：

```
// 检测BeforeGap的有效性并试图合并Prev Block
function FillerSizeBeforeGap(a: PChar): Integer;

// 检测AfterGap的有效性并试图合并Next Block
function FillerSizeAfterGap(a: PChar): Integer;
```

除了检测合并的有效性之外，这两个例程也试图根据 BeforeGap 和 AfterGap 的 sizeFlags 域中的长度和标志信息来定位 Prev Block 和 Next Block。如果这些块是空闲的，那么这两个例程也将会进一步向前、向后合并。

调用这两个例程后并修正过 addr 和 size 值之后，块 b 的起始地址被赋值给 curAlloc，而长度值记录在 remBytes 中。至此，一块新的物理存储提交完成，它的首地址总是记录在 curAlloc 中。这存在三种可能：

- 调用 MergeBlockAfter()时未合并 BeforeGap，则 curAlloc 指向 b.addr；
- 调用 FillerSizeBeforeGap()时未合并 Prev Block，则指向 merged.addr；
- 调用 FillerSizeBeforeGap()时发生了向前合并，则指向 Prev Block。

回到 TryHarder()。由于 curAlloc 指向新分配的物理存储，因此在调用 NewCommit()后，紧接着访问 curAlloc，并从前面截取指定长度的块即可。而后续的代码，则用于处理从 avail 链表与 smallTab 中切分内存块的情况。

■ 内存再分配(ReallocMem)

内存再分配，实际上是指对一个在用的内存块进行长度调整。根据这个内存块在内存中的后续块的情况，可能有以下处理方案：

- 如果新长度小于原内存长度，则截去原内存的尾端部分，并试图回收；
- 如果新长度大于原内存长度，且后续内存块空闲，则合并后续内存块；
- 如果新长度大于原内存长度，但没有足够后续空闲内存块，则调用重新分配内存，并将原内存块中的内存复制到新内存块中。

很显然，无论新长度的大小如何，需要实现内存再分配的最简单的方法是这样的：

```
p := SysGetMem(size);
oldSize := (PUsed(PChar(p)-sizeof(PUsed)).sizeFlags and not cPFlags) -
  sizeof(TUsed);
if oldSize > size then
  oldSize := size;
if n <> nil then begin
  Move(p^, n^, oldSize);
  SysFreeMem(p);
end;
result := n;
```

不过，这在 SysReallocMem()中是备选的方案。对于内存再分配来说，最消耗时间的操作是：

- 重新提交内存的 NewCommit()操作；
- 重新分配内存的 SysGetMem()操作；
- 复制原有数据的 Move()操作。

而上面的代码就占了其中两项，因此效率很低。为此，SysReallocMem()中引入了一个 ResizeInPlace()例程，它用于在指针的原有位置上，通过合并和切分内存块来实现重分配。

```
// ReallocMem()例程
function SysReallocMem(p: Pointer; size: Integer): Pointer;

// 通过截去多余内存或合并后续空闲内存块来实现内存再分配
function ResizeInPlace(p: Pointer; newSize: Integer): Boolean;
```

对于缩小空间的操作，ResizeInPlace()例程主要实现的操作是回收多余的内存空间。如果这些空间在 curAlloc 指向的负偏移处，则简单地调整 curAlloc 的指针即可；否则，先将这些空间标注为“在用的”，然后调用 InternalFreeMem()例程回收。

对于增大空间的操作来说，ResizeInPlace()则主要通过合并后续(连续)空闲内存块来获取足够的空间。如果发现后续内存块是 Gap 记录，则调用 NewCommitAt()来提交 Gap 记录后的物理存储。

仅在指定内存块后没有足够的连续空闲内存，且没有 Gap 记录(或者 Gap 记录后的能提交的空间小于请求的长度)时，ResizeInPlace()例程才返回 False，通知 SysReallocMem()使用前面所述的备选方案来实现内存再分配。

7.5.5　初始化、结束化与其他辅助例程

Delphi 内部的内存分配机制由以下两个例程来实现初始化和结束化：

```
function InitAllocator: Boolean;
procedure UninitAllocator;
```

缺省情况下，Delphi 并不初始化内部内存管理器。而是在第一次调用 GetMem()、FreeMem()或 ReallocMem()例程时，实时检测初始化状态 initialized，并调用 InitAllocator()例程。因此内核中只需要将 UninitAllocator()被加在 *System.pas* 的单元结束化节。

在 *GetMem.inc* 中，被公开的例程还有 GetHeapStatus()，这是相关的例程代码：

```
// 双向环形链表中的地址空间(TBlockDesc.Size)的和
function BlockSum(root: PBlockDesc): Integer;

// 获取内存堆状态信息记录
function GetHeapStatus: THeapStatus;
```

GetHeapStatus()用于反馈 Delphi 的内存堆的状态，这是一个 THeapStatus 记录。主

要的信息包括：

- TotalAddrSpace：spaceRoot 链表中，全部保留的地址空间大小；
- TotalUncommitted：decommittedRoot 链表中，全部的未提交内存大小；
- TotalCommitted：committedRoot 链表中，全部的已提交内存大小；
- TotalAllocated：用户代码通过 GetMem() 分配的内存总长度；
- TotalFree：FreeSmall + FreeBig + Unused；
- FreeSmall：smallTab 中所有空闲块的大小；
- FreeBig：avail 中所有空闲块的大小；
- Unused：curAlloc 指针所指向的内存块的大小；
- Overhead：所有 Gap 和在用内存块头部信息的总长度。

GetHeapStatus() 中提供了一种遍历全部已提交内存块(即 committedRoot 链表)的方法，这事实上是对内存管理器的一些细节的总结。在下一节“遍历全部内存块”中也将使用类似的技术，源于 GetHeapStatus() 的安全性，这些技术也将是安全的。

7.6 遍历全部内存块

在讨论内存管理器的替换时，曾经特别指出：由于新的内存管理器不能区分哪些内存块是原内存管理器分配的，所以应该保证替换前 AllocMemSize=0。

Delphi 的内存管理器并不提供任何例程来识别一个内存块指针是否归该内存管理器管理。这使得替换内存管理器惟一安全的方法是：将新的内存管理器单元放在项目文件的 uses 列表的头部。

编写一些调试、测试和分析软件时，可能需要列举每一个内存块并分析。因此，和编写第三方内存管理器一样，了解如何遍历 Delphi 缺省内存管理器的全部内存块，是一个必须克服的技术壁垒。

现在已经确知内存管理器包含了如下的一些结构：

- blockDescBlockList: PBlockDescBlock;
- blockDescFreeList: PBlockDesc;
- spaceRoot: TBlockDesc;
- decommittedRoot: TBlockDesc;
- committedRoot: TBlockDesc;
- smallTab: ^TSmallTab;
- avail: TFree;

- curAlloc: PChar;

一个块描述(TBlockDesc)总是存在于 blockDescBlockList 指向的结构中，而这个结构是在当前进程的堆上分配内存的，因此可以遍历进程的堆来找到全部 TBlockDescBlock 结构。接下来：

- 依据 TBlockDescBlock.Next 的关系，可以找到单向链表的第一个结点；
- 收集 TBlockDescBlock.Data 域，可得到包含全部 TBlockDesc 的列表。

一个块描述要么是未用的，要么就存在于 spaceRoot、committedRoot 和 decommittedRoot 三个环形链表之中。如果块描述未用，则它存在于 blockDescFreeList 指向的单向链表中。

如果从环形链表中的任何一个结点开始依序访问，那么最终会回到开始结点；而单向链表最终会找到一个 nil。因此，根据这个特点，可以将块描述分别列在一个单向链表和三个环形链表中。能确知的是，这个单向链表就是 blockDescFreeList。

接下来讨论如何区分三个环形链表。

首先，如果已确知一个已分配内存块的指针，那么这个指针必然不会存在于 decommittedRoot 中。因此，只需要用 GetMem()来获取一个内存块，并引入它的指针到三个环形链表中验证一下就可以了。

接下来，如果有一个存在于 decommittedRoot 中的无效指针，那么它必须存在于 spaceRoot 中（当然，它不会存在于 committedRoot 中）。因此，只需要从 decommittedRoot 所管理的地址空间中任意取一个指针，并引入到剩下的两个环形链表中验证，就可以找到 spaceRoot。

这里有一种特殊情况。如果所有的保留地址空间都被提交，即 decommittedRoot 链表为空，那么又如何区分 spaceRoot 与 committedRoot 呢？

在这种情况下，committedRoot 和 spaceRoot 所表示的空间是完全重叠的，尽管所包含的块描述(TBlockDesc)数量、记录域值存在不同，但并不会影响到下一步对内存块的遍历。

最后，在一个已经提交的内存存储中存在的内存块，要么是空闲的(TFree)，要么是在用的(TUsed)，或者就是 Gap 记录。而在 committedRoot 链表中遍历这些内存块的方法，在例程 GetHeapStatus()中有完整的实现代码。

由于在 committedRoot 中可以列举出全部的空闲内存块，因此是否查找剩下的一些结构

(例如 smallTab、avail 和 curAlloc)并不重要。但事实上也并不复杂：

- smallTab 是比较容易找到的，它也是在进程堆上进行分配的，其大小为 sizeof(TSmallTab)。因此也可以通过列举堆来查找。
- curAlloc 要么是提交空间的第一个块，要么就一定附在一个 Used 块之后，而且它一定是一个无效的块(没有有效的 TFree 头部)。
- 在 committedRoot 中，既不在 smallTab 中，且不是 curAlloc 的空闲内存块，就是 avail 链表中的一个结点。由于 avail 也是一个双向环形链表，所以只要遍历该链表，就可以找到 avail 指针(avail.size=0)。

上述思路的完整实现代码，请参阅随书光盘中的示例程序 MemMapper。

7.7 共享内存管理器

共享内存管理器是 Delphi 内存管理机制的一个重要组成部分。共享内存管理器由 *ShareMem.pas* 和 *BorIndmm.dll* 两个部分组成。

Delphi 的官方文档中指出：如果需要在 DLL 中导出例程，需要传递长字符串或动态数组参数，或这些类型的返回值(包括在记录和对象中使用它们)，那么需要使用到 *borindmm.dll* 和 *ShareMem.pas* 单元。

这是因为 Delphi 中的长字符串、动态数组、变体、接口等都是使用引用计数机制来进行管理的。此外，在数组的元素、记录的域和类的字段中使用上述类型，也将会间接地与引用计数机制发生关系。

引用计数使得一块内存占用可能同时被多个变量引用，只有在所有变量都不再使用这块内存时，它才会被释放。释放这个内存块的操作，是在最后一个引用它的变量被释放的同时发生的。

然而，对于 Delphi 的编译器来说，它并不能识别一个参数是否是从 DLL 传递到 HOST 中来的(或者反之)，所以编译器也总是对它(返回值或入口参数)加入管理引用计数的代码，并按照同样的引用计数机制来管理。这样，一个在 DLL 中分配的内存块，可能在 HOST 中因为失去引用而被释放。例如：

```
// in dll
function GetStringFromDLL(Str: String) : String;
begin
  Result := Str + ' Sub String From Dll.';
```

```
end;

// in host
function GetString : String;
begin
  Result := GetStringFromDLL('Shared String.');
end;

// call GetString() from host app.
begin
  writeln(GetString);
end.
```

在 writeln()调用完成后，GetString()返回的字符串不再使用，所以 Delphi 编译器将加入一个_LStrClr()的调用来完成引用撤消(使引用计数减 1)。而_LStrClr()完成引用撤消之后，又会因为引用计数减至 0，调用 FreeMem()来释放内存——这时，HOST 中试图释放在 DLL 中创建的字符串。因此，接下来的情况是：引发一个无效的指针操作（EInvalidPointer 异常）。

但这并不是问题的全部。因为即使是在 DLL 中传出一个 GetMem()获取的内存指针，并在 HOST 中释放它(或者反之)，也会有类似的错误。所以，问题真正的根源是在于“DLL 和 HOST 分别使用了不同的内存管理器”，至于在长字符串等数据类型中使用引用计数，不过是使得“在哪个模块中释放内存”变得更加不可预测而已。

Borlndmm.dll 可以解决上述问题。

Borlndmm.dll 的代码其实非常简单，我想是简单到了 Borland 无法公开这些代码的地步。如果要复原 *Borlndmm.dll* 的代码，大概只需要这样几行就可以了：

```
library Borlndmm;

// UNKNOWN
// procedure HeapAddRef;
// procedure HeapRelease;
procedure DumpBlocks;
begin
  // UNKNOWN
end;

function GetAllocMemCount: Integer;
begin
  Result := System.AllocMemCount;
end;
```

Delphi 源代码分析

```
function GetAllocMemSize: Integer;
begin
  Result := System.AllocMemSize;
end;

exports
//  HeapAddRef name '@Borlndmm@HeapAddRef$qqrv',
//  HeapRelease name '@Borlndmm@HeapRelease$qqrv',
  DumpBlocks, GetHeapStatus, SysGetMem, SysFreeMem, SysReallocMem,
  System.SysGetMem name '@Borlndmm@SysGetMem$qqri',
  System.SysFreeMem name '@Borlndmm@SysFreeMem$qqrpv',
  System.SysReallocMem name '@Borlndmm@SysReallocMem$qqrpvi';

begin
  System.IsMultiThread := True;
end.
```

从原理上来说，只要有一个共用的 DLL 在其他 DLL 模块和 HOST 进程载入之前加载，并保证所有 DLL 与 HOST 进程都能使用共用 DLL(例如 *Borlndmm.dll*)中的内存分配例程，那么这个 DLL 就可以用来做共享内存管理器。

ShareMem.pas 并不包括实现内存分配例程的代码，它主要用来保证当前模块会静态装载 *Borlndmm.dll*。当然，它也包含为数不多的几行代码来调用 `SetMemoryManager()`，以使用 *Borlndmm.dll* 中的例程去替换缺省的内存管理器。

由于在每一个.DLL 或.EXE 中都引用了 *ShareMem.pas* 单元，所以 *Borlndmm.dll* 模块一定会比这些.DLL 和.EXE 更先载入内存，此后任何启动的.DLL 或.EXE 都将使用 *Borlndmm.dll* 模块中的内存管理器代码。由于无法确定后续模块中是否使用了多线程，因此 *Borlndmm.dll* 总是直接初始化 `IsMultiThread` 为 `True`——关于这一点的真实细节仍不是非常清楚，也可能 *Borlndmm.dll* 中使用独立的 *GetMem.inc* 文件，而其中的代码并不检测 `IsMultiThread` 状态，就直接进入临界区。

直接使用临界区是导致 *Borlndmm.dll* 的内存管理器性能下降的罪魁祸首(极端的情况下，它速度只有缺省的内存管理器速度的 1/8)。除了在临界区使用上的细小的区别之外，*Borlndmm.dll* 中使用的代码与 *GetMem.inc* 完全一致。这也是上例中可以直接导出 *System.pas* 单元中的例程的原因。

7.8　第三方内存管理器

第三方的内存管理器是不多见的，其中包括：

- QStrings 的作者 Andrew Driazgov 发布的 QMemory 和 ShareQmm 2.01a。
- High Performance Delphi 发布的 HPMM。

其中，HPMM 尽管提出了较高性能的优化算法，却并没有实现 ShareMem 的接口。因此，HPMM 仅能在单独模块中使用(MultiMM 单元用于多线程环境)。

ShareQmm 实现共享内存管理器的思路与 Delphi 是一致的：它用一个 *Qmm.dll* 来替代 *BorIndmm.dll*。因此，使用 ShareQmm 时仍然得背上 *Qmm.dll* 的包袱。更加危险的是：没有太多人知道 *Qmm.dll* 的用途，因此被从硬盘上意外清除的可能性远比 *BorIndmm.dll* 大。

也有一些其他的专门用来替换 *BorIndmm.dll* 和 *ShareMem.pas* 的共享内存管理器。例如使用“内存桥”技术的 FastShareMem 和 ShareMemRep，它们都不须要另外加载模块，就能在 DLL 和 HOST 之间共享内存管理器。

FastShareMem 使用 API 在虚地址空间的高端，查找一个 64K 的页，然后提交它，并将一个当前的内存管理器的记录写到该地址上。在其他模块中初始化时，就可以通过列举内存页查找到这个记录，并使用它来替换当前的内存管理器。

ShareMemRep 是在 FastShareMem 的基础上发展起来的另一个 OpenSoruce 项目。不同的是，它通过导出表在 EXE 和 DLL 之间传递内存管理器地址(这个地址在编译期就被写入到导出表了)。除了内存管理器的地址之外，ShareMemRep 还导出了一个 SharedModules 记录，用来记录、管理所有被“桥接”的动态和静态模块。此外，ShareMemRep 还实现了内存管理器与其他模块间实现了 `IsMutilThread` 状态的同步，这表明 ShareMemRep 在多线程状态下也是安全的。

一旦 DLL 与 HOST 成功地建立了内存管理的“桥接”关系，那么 *BorIndmm.dll* 就不再是必需的了。因此，“内存桥”的技术核心，在于将内存管理器地址记录到某个位置，以供其他模块查找和替换。

关于 ShareMemRep 的详细实现代码及其实现机制，请参阅随书光盘中的 ShareMemRep 1.5 以及《共享 HOST 与 DLL 之间的内存管理器》一文。表 7-1 仅列出了 ShareMemRep 与 FastShareMem 之间的一些差异。

表 7-1 ShareMemRep 与 FastShareMem 之间的一些差异

解决方案支持	解决方案(或程度)		备注
	FastShareMem	ShareMemRep	
共享MemMgr	记录在内存高端	通过导出表导出内存管理器地址	
多线程	不支持	可同步DLL与HOST的多线程状态	
模块识别	不能区分动态和静态模块	构造模块表，可识别静态和动态DLL载入	
静态DLL	可在静态DLL中共享MemMgr	可在静态DLL中载入MemMgr	
动态DLL	不能阻止共享其MemMgr	动态DLL不能共享它的MemMgr	
未卸载库的检测	通过引用计数识别	通过模块表识别	
异常	支持	支持	
第三方MemMgr	必须加入到第一个载入模块中	可以加入到静态模块和HOST模块中	①
替换的完整性	不完整	完整地实现了ShareMem.pas的全部接口	

7.9 小结

块描述来记录被保留的虚地址空间(spaceRoot)，以及虚地址空间中已提交(commitRoot)和未提交(decommitRoot)的部分。每次提交和取消提交都进行页边界对齐运算。

TBlockDescBlock 结构用于每次申请 100 个块描述结点。空闲的块描述被回收到 blockDescFreeList，这是一个后入先出队列(LIFO)。

被保留的虚地址空间称为“堆空间”，在堆中可能存在的块包括：已提交物理存储和未提交物理存储。如果堆空间用尽(未提交物理存储用尽)，则再保留新的虚地址空间。

已提交物理存储即是可用的内存块，包括在用、空闲和 Gap 三种状态。

在用内存块结构较为简单，使用 TUsed 记录来描述；空闲内存块结构复杂，在首尾两端存在 TFree 记录。

没有结构用于登记在用内存块，仅在 AllocMemCount 和 AllocMemSize 中保存了它们的数量和总长度。空闲的可用内存块则总是被登记在：

① Delphi 的 ShareMem 不支持第三方内存管理器。

- smallTab 数组各元素所指向的双向环形链表；
- avail 双向环形链表；
- curAlloc 指针指向的内存块；

上述三者之一中。

已提交的物理存储，在其前后内存边界上会填充 Gap。Gap 是 cFillerFlag 为 1 的 TUsed 记录，它存在的目的，是用来标识该物理存储相邻的地址空间是未提交的，不能合并。

堆空间中的合并操作是指块描述所记录的地址空间的合并。内存块的合并操作则指的是通过 TFree 和 TUsed 记录实现的向前、向后合并。

只有在首次调用内存管理例程时才会初始化内存管理器，所以在此前替换内存管理器时并不会留下任何内存垃圾。正是因此，*ShareMem.pas* 和 *BorIndmm.dll* 在“节省内存”方面有很好的表现。但是由于 *BorIndmm.dll* 缺省时是按多线程方式来运行的，所以它的效率会差得很多。

ShareMem.pas 和 *BorIndmm.dll* 是为了实现“HOST 与 DLL 模块共用一个内存管理器”。FastShareMem 和 ShareMemRep 使用内存桥技术实现了相同的目的。与“抛弃 *BorIndmm.dll*”所带来的优点相对应的是：可能必须在一些条件限制下使用它们。

第8章

错误和异常

Delphi 内部有两套出错处理机制：错误和异常[①]。这其中的异常又有两种实现方式：操作系统异常处理和 Delphi 异常处理。

Delphi 的异常处理机制是依赖于操作系统的结构化异常处理机制的。准确地说，Delphi 异常处理是对操作系统的异常处理机制的一个具体实现。

SysUtils.pas 在 Delphi 的异常处理机制中占据了很重要的位置。是否引用 *SysUtils.pas* 单元，对异常的俘获、处理都有影响。本章将详述个中缘由。

本章将涉及到异常对象的创建、传递和使用机制，但并不描述类或者对象本身的机制。

8.1 错误

Delphi 内核中的“错误”，通常是代码中能够被检测到的，例如数组或字符串的边界错误、无效的内存指针以及 IO 错误一类的错误。它们通常以类似如下的代码调用到 Error() 例程：

```
function _GetMem(Size: Integer): Pointer;
begin
  if Size > 0 then
  begin
```

① 断言(Assert)在 *System.pas* 中被理解为错误处理中的一种。

```
  Result := MemoryManager.GetMem(Size);
  if Result = nil then          // 检测
    Error(reOutOfMemory);       // 调用错误处理
// ...
```

调用 Error()例程时传入的错误代码 errorCode 将被传送入 ErrorAt()例程。与此相关的数据还有[ESP]:基于堆栈处理的原则，调用 Error()时，栈顶的值正好是 ErrorAt()后返回的地址，也就是出错代码的下一行。

```
procedure Error(errorCode: TRuntimeError);
asm
        AND     EAX,127
        MOV     EDX,[ESP]
        JMP     ErrorAt
end;

procedure ErrorAt(ErrorCode: Byte; ErrorAddr: Pointer);
const
  reMap: array [TRunTimeError] of Byte = (
  // ...
  );
begin
  errorCode := errorCode and 127;
  if Assigned(ErrorProc) then
    ErrorProc(errorCode, ErrorAddr);
  if errorCode = 0 then
    errorCode := InOutRes
  else if errorCode <= Byte(High(TRuntimeError)) then
    errorCode := reMap[TRunTimeError(errorCode)];
  RunErrorAt(errorCode, ErrorAddr);
end;
```

在 Error()的后续处理中，系统将调用到_Halt0()退出，_Halt0()中的如下代码用于生成并输出一个错误消息串，这是内核中仅有的“错误显示”的地方：

```
procedure RunErrorAt(ErrCode: Integer; ErrorAtAddr: Pointer);
begin
  ErrorAddr := ErrorAtAddr;
  _Halt(ErrCode);
end;

procedure _Halt0;
begin
  // ...
  if ErrorAddr <> nil then
```

```
  begin
    MakeErrorMessage;
    WriteErrorMessage;
    ErrorAddr := nil;
  end;
// ...
```

系统全局变量 ErrorProc 可以用来设置错误处理例程。错误处理中，不能像异常处理那样，返回到执行点之后的语句继续执行。因为 ErrorProc 变量所指向的例程执行完成之后，系统仍然会通过 RunErrorAt()调用_halt0()例程来中止程序。

的确可以有一些技巧来使 ErrorProc 处理完后退回到错误发生点之后。但这绝对不是好主意：处理一个隐藏的错误带来的危害，通常高于解决错误本身所花的成本。如果执意要这样做，下面的代码可以提供一个参考：

```
procedure LogError(ErrorCode: Integer; ErrorAddr: Pointer);
begin
  writeln('Code: ', ErrorCode, ' Addr: ', LongInt(ErrorAddr));

  // 清除堆栈，返回到Error()例程调用之前①
  asm
    {$IFOPT W+}
      RET 6 * TYPE DWORD
    {$ELSE}
      RET 3 * TYPE DWORD
    {$ENDIF}
  end;
end;

var
  p : Pointer;
begin
  ErrorProc := @LogError;

  getmem(p, 1024*1024*1024);  // 试图分配过大的内存块，引发 reOutOfMemory 错误
  readln;
end.
```

一些情况下，系统并不会给用户代码以处理 ErrorProc 变量的机会。例如用户代码调用

① 这里对{$W}编译开关进行检测的原因在于，如果使用{$W+}开关编译，会使编译器在异常帧之前压入 3 个 DWORD 的堆栈帧。这通常用于一些调试器检测例程相互调用的层次关系。

纯虚方法(Abstract Method)时发生的 AbstractError，就直接调用例程 RunError()来退出进程。这个例程与 RunErrorAt()仅有小小的不同：

```
procedure       _AbstractError;
asm
        CMP     AbstractErrorProc, 0
        JE      @@NoAbstErrProc
        CALL    AbstractErrorProc

@@NoAbstErrProc:
        MOV     EAX,210
        JMP     _RunError
end;

procedure _RunError(errorCode: Byte);
asm
        POP     ErrorAddr
        JMP     _Halt
end;
```

8.2　断言

断言的例程声明是：

```
procedure Assert(expr : Boolean [; const msg: string]);
```

但它在 *System.pas* 中却是这样来实现的(PUREPASCAL 版本并不能得到有效的 ErrorAddr 参数)：

```
procedure _Assert(const Message, Filename: AnsiString;
  LineNumber: Integer);
{$IFDEF PUREPASCAL}
begin
  if Assigned(AssertErrorProc) then
    AssertErrorProc(Message, Filename, LineNumber, Pointer(-1))
  else
    Error(reAssertionFailed);  // loses return address
end;
```

```
{$ELSE}
asm
        PUSH    EBX
        CMP     AssertErrorProc,0
        JNZ     @@1
        MOV     AL,reAssertionFailed
        CALL    Error
        JMP     @@exit

@@1:    PUSH    [ESP+4].Pointer
        CALL    AssertErrorProc

@@exit:
        POP     EBX
end;
{$ENDIF}
```

在缺省状况下(不引用 *SysUtils.pas* 单元时)，AssertErrorProc 是没有初值的，这表明此时 Delphi 按照与普通的“错误”一致的规则来处理“断言”。也就是说，当断言条件为 False 时，将直接调用 Error()例程，并中止应用程序的运行。

但是，给 AssertErrorProc 例程赋值后，情况却与给 ErrorProc 赋值不同。由于在 AssertErrorProc 调用后没有其他的代码，所以将很好地维护调用 Assert()之前的程序执行状态。

这样的规则为使用 Assert()进行错误报告提供了足够的便利。

Assert()语句是否被编译到代码中，可以用$ASSERTIONS 编译选项来设定，也可以在 IDE 中的 Project Options 中方便地设定它。一旦{$ASSERTIONS OFF}，那么目标文件中将不会有对 Assert()例程的调用。

在应用中程序中加入下面的单元，可以通过控制 ASSERTIONS 状态来切换系统错误报告，而且这种报告比一些普通的 Log 系统要更为详细：

```
unit AssertLogs;

interface

implementation
uses
  Windows,
  SysUtils;
```

```
var
  runErrMsg : String;
  oldAssertErrorProc : TAssertErrorProc;

procedure LogAssert(const Message, Filename: string; LineNumber: Integer;
  ErrorAddr: Pointer);
begin
  runErrMsg := format('Error: %s, Addr: %p, in file(%d): %s',
    [Message, ErrorAddr, LineNumber, FileName]);
  if IsConsole then
    Writeln(runErrMsg)
  else
    MessageBox(0, runErrMsg, 'Error Log by AssertLogs', 0);
end;

initialization
  oldAssertErrorProc := AssertErrorProc;
  AssertErrorProc := @LogAssert;

finalization
  AssertErrorProc := oldAssertErrorProc;
end.
```

使用这个单元时，用户的应用程序中，原来定义错误消息输出时的代码如下：

```
{$IFDEF Debug}
  Debugger.LogMsg('CompilerPath: ' + FCompilerPath);
  Debugger.LogMsg('CompilerRegPath: ' + FCompilerRegPath);
{$ENDIF Debug}

// or

{$IFOPT D+}
  Debugger.LogMsg('输出调试信息!');
{$ENDIF}
```

就可以直接写成：

```
Assert(false, 'CompilerPath: ' + FCompilerPath);
Assert(false, 'CompilerRegPath: ' + FCompilerRegPath);
Assert(false, '输出调试信息!');
```

在需要输出调试信息或者给测试用户发布 Beta 时，可以用{$ASSERTIONS ON}编译。在正式发布时，用{$ASSERTIONS OFF}编译即可。

在 *AssertLogs.pas* 中的 procedure LogAssert()，只是一个实现框架，可以编写更加复杂的记录或者报告处理例程。

使用 *AssertLogs.pas* 例程的另一个好处，是不需要第三方控件和调试信息的支持，仍然可以在 Log 中得到 LineNumber 信息。这是因为，在使用 AssertLog.pas 编译的目标文件中，行号信息是编译器直接传入 Assert() 例程的。而类似于 ExceptionalMagic 和 EurekaLog 等第三方的工具，都是在通过读取调试信息而得到行号等数据的。

8.3 Windows 与 Delphi 中的异常处理机制概要

Windows 使用被称为“结构化异常处理”(SEH，Structured Exception Handling)的机制，使得开发人员可以得到处理异常和错误的机会。Delphi 将这种机制的具体实现，内置到下列位置：

- 编译器：实现了 SEH 的基本框架，并在例程的初始化、结束化中自动加入异常处理代码。
- *System.pas* 单元：实现了 except、finally 两种类型的异常处理框架，为必要的代码加入异常帧(Exception Frame)；实现了 finally 型的异常处理。
- *SysUtils.pas* 单元：实现了 except 型的异常处理，实现了异常类。

与通常理解的“异常”不同的是，SEH 实际包含了下面三个部分：

- 结束处理：**try-finally** 结构的处理称为结束处理。它与被保护块是否引发异常并没有直接关系。由 **finally-end** 关键字定义的代码总是能够得到执行[1]。
- 异常处理：**try-except** 结构的处理称为异常处理。如果被保护块引发异常，那么将会进入 **except-end** 关键字定义的代码块。一旦异常得到处理，且没有重新引发(raise)异常，则程序继续执行 **except-end** 后的代码。
- 顶层(top-level)异常处理[2]：通过 API：UnhandledExceptionFilter() 可以设定一个顶层异常处理例程，它被置入到进程和该程序的全部(已经创建和将会创建的)线程的顶部。亦即是说，顶层异常处理是应用程序(的线程或进程)最后一次捕获到异常

(1) 有一些特殊的情况，会导致 finally-end 块并不能总是得到执行。例如使用 ExitThread()、TerminateThread()、ExitProcess()、TerminateProcess() 等来使得线程或进程退出、中止。此外，如果线程在 try-finally 块中被锁死，那么 finally-end 也不会得到执行。

(2) 尽管“顶层异常处理”也是“异常处理”的一部分，但它并不通过 try-except-end 结构来指定，因此在本书中被单列出来

时的处理代码。

Windows 的 SEH 按照下面这样一种顺序，查找并处理异常(这并不包括“结束处理”)：

- 调试器：异常被先送交到调试器处理。如果没有调试器，或调试器不处理，或者处理完后指定交给其他异常处理例程，则，
- 线程相关的异常处理例程[①]：应用程序可以安装线程相关的异常处理例程，异常会由操作系统自动送交给它处理。线程相关的异常处理可以是一个链式结构，这种情况下，一个异常可以被多个处理例程连续处理。如果没有安装，或者处理完后指定提交给其他异常处理例程，则：
- 调试器：如果链上的所有线程相关处理例程都不处理该异常，且应用程序还处于调试状态，则再次交回到调试器处理。如果应用程序不处于调试状态，或调试器也不处理该异常，则，
- 顶层异常处理：检测是否安装了顶层异常处理例程，并转向对它的调用。如果没有顶层异常处理例程，或它不处理这个异常，则，
- 默认系统处理程序：这是最后的一些处理。通常显示一个对话框，可以选择关闭或者最后将其附加到调试器上。

这个流程的前两步，在 Delphi IDE 中也能被清晰地体现出来。以下面的代码为例：

```
procedure TestExceptionForIDE;
begin
  try
    format('Make Exception: %d', ['string']);
  except
    on e:EZeroDivide do;
  end;
end;
```

首先，对于应用程序来说，Delphi IDE 也是调试器。如果打开“Debugger Option\Language Exceptions\Stop On Delph Exceptions”选项，则在 try-except/finally-end 中发生的异常会导致 IDE 弹出一个“Debugger Exception Notification”提示框。因此，这时会看 IDE 出现如下的对话框（图 8.1）：

① “异常处理”和“结束处理”是高级语言对“线程相关的异常处理”的两种实现方式。参见下一节中的“从操作系统的角度来理解编译器行为”。

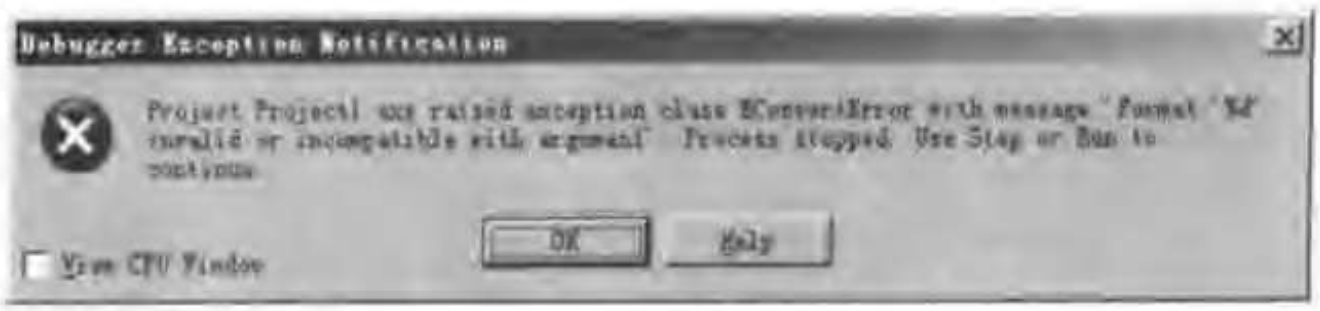


图 8.1

在点击[OK]按钮后，调试指针会指向例程中 format()调用的语句。

接下来，异常交由 except-end 块中的代码来处理。

按照操作系统的约定，如果异常保护块中的代码没有处理异常(且上一层没有其他的异常处理代码)，那么异常还会再次发给调试器和使用 SetUnhandledExceptionFilter()安装的顶层异常处理例程(第三、四步)。

但是这种情况对 Delphi 的用户来说，却是很难看到的了。因为 Delphi 内置调试器在第二次得到异常时，常见的处理方法就是直接中止进程(这会有很多的例外，在后面的小节中将一一补述)。调试如下示例代码时，进程很快就中止了，而 IDE 看起来像是没有捕获到异常一样：

```
program TestExceptionForDebugger;
{$APPTYPE CONSOLE}

var
  p : procedure;
begin
  p;  // 调用一个指向nil的过程，引发异常
end.
```

而事实上，IDE 中的内置调试器是在第二次得到异常时，中止了进程。

Delphi 并不缺省为用户调用 SetUnhandledExceptionFilter()安装一个顶层处理例程。因此，如果发生了不可预知，且 Delphi 内核也无法识别或处理的异常，那么最后的一道防线只会是“默认系统处理程序”(第五步)。如果到达这里，用户可能的选择只能是启用一个其他的调试器，或者中止应用程序。仍以上例为例，如果不是在调试环境(IDE)中运行代码的编译文件，那么会立即得到如图 8.2 所示的提示框：



图 8.2

8.4　编译器对异常处理机制的实现

绝大多数情况下，实现 SEH 是编译器的事。在很多不可见的地方，Delphi 会自动加入异常处理的代码。而且几乎所有使用 try-finally/except-end 结构的地方，编译器都直接加入了代码——重要的是，有些代码在 *System.pas* 中根本就不存在。

研究编译器对 SEH 的实现是件很难的事。但是不用担心，有了 Delphi IDE 中的调试器和一些有 SEH 的基本知识，一切都可以迎刃而解。

8.4.1　最小化内核的启示

Delphi 最小化内核中，至少必须存在三个例程：

```
procedure _InitExe;
procedure _HandleFinally;
procedure _Halt0;
```

其中_InitExe()和_Halt0()都容易理解，它们决定了一个应用程序的开始和结束。但为什么_HandleFinally()必须存在呢？

_HandleFinally()用于实现“结束处理”(下文中将直接称为 finally 型处理)。而 finally 型处理并不是与异常直接相关的，它更重要的是用来保护代码块。

简要地说，finally 型处理用来保障一些代码“无论如何”都将被执行。但，哪些代码有如此的重要性呢？下例说明这一点：

```
type
  TRec = record
    I : Integer;
    S : String;
  end;

procedure TestRecordAddRef(R : TRec);
begin
  R.S := '';
end;
```

这个例子在第 2 章中的“增加引用何时发生”一节中曾引用过，以前的内容曾经指出，在 TestRecordAddRef()例程的 begin 中，会加入对_AddRefRecord()调用。那么很显然的：

在 end 行中还应该会加入对_FinalizeRecord()的调用来使引用撤消。

但是，如果在 begin..end 之间的代码行执行发生了异常，就将导致例程异常退出(到调用者例程)。此时如果无法执行到_FinalizeRecord()，则将导致对 TRec 记录中的 S 字符串引用无法撤消。

为了使得_FinalizeRecord()总是被执行到，编译器在 TestRecordAddRef()的初始化和结束化代码中加入了 finally 型处理，在这里就可以看到对_HandleFinally()的调用。反编译代码如下：

```
Unit1.pas.35: procedure TestRecordAddRef(R : TRec);
Unit1.pas.36: begin
0044C9BC 55              push ebp
0044C9BD 8BEC            mov ebp,esp      // 在栈顶建立一个8字节的R记录的备份
0044C9BF 83C4F8          add esp,-$08
0044C9C2 56              push esi
0044C9C3 57              push edi
0044C9C4 8BF0            mov esi,eax
0044C9C6 8D7DF8          lea edi,[ebp-$08]
0044C9C9 A5              movsd
0044C9CA A5              movsd

//以下操作针对备份[ebp-$08]进行,记为: _R
0044C9CB 8D45F8          lea eax,[ebp-$08]
0044C9CE 8B15A0C94400    mov edx,[$0044c9a0]

//_AddRefRecord(_R, TypeInfo(TRec));
0044C9D4 E8CB7FFBFF      call @AddRefRecord
0044C9D9 33C0            xor eax,eax                  // 1). eax := 0
0044C9DB 55              push ebp                     // 2). 压入栈基地址ebp
0044C9DC 680BCA4400      push $0044ca0b               // 3). 压入异常描述记录指针
// 4). 压入上一个异常帧地址(FS:[0])
0044C9E1 64FF30          push dword ptr fs:[eax]
// 5). 将新异常帧地址(栈顶)送到FS:[0]
0044C9E4 648920          mov fs:[eax],esp
Unit1.pas.37: R.S := '';
0044C9E7 8D45FC          lea eax,[ebp-$04]
0044C9EA E82174FBFF      call @LStrClr                // R.S := '';
0044C9EF 33C0            xor eax,eax
0044C9F1 5A              pop edx                      // 开始清除栈顶异常帧
0044C9F2 59              pop ecx
0044C9F3 59              pop ecx
0044C9F4 648910          mov fs:[eax],edx
0044C9F7 6812CA4400      push $0044ca12               // 为下面的ret语句准备返回地址
```

```
0044C9FC 8D45F8          lea eax,[ebp-$08]
0044C9FF BB15A0C94400    mov edx,[$0044c9a0]

// _FinalizeRecord(_R, TypeInfo(TRec));
0044CA05 E8567EFBFF      call @FinalizeRecord
0044CA0A C3              ret                          // 返回到 $0044ca12
0044CA0B E9046EFBFF      jmp @HandleFinally
0044CA10 EBEA            jmp -$16
Unit1.pas.38: end;
0044CA12 5F              pop edi                      // 恢复edi
0044CA13 5E              pop esi                      // 恢复esi
0044CA14 59              pop ecx                      // 清除 _R 的四字节(共8字节)
0044CA15 59              pop ecx                      // 清除 _R 的四字节(共8字节)
0044CA16 5D              pop ebp                      // 恢复ebp
0044CA17 C3              ret
```

在 begin 语句中，下面的代码：

```
0044C9DB 55              push ebp                     // 2), 压入栈基地址ebp
0044C9DC 680BCA4400      push $0044ca0b               // 3), 压入异常描述记录指针
// 4), 压入上一个异常帧地址(FS:[0])
0044C9E1 64FF30          push dword ptr fs:[eax]
```

用于形成一个异常帧记录 TExcFrame。其结构在 *System.pas* 被定义为：

```
type
  PExcFrame = ^TExcFrame;
  TExcFrame = record
    next: PExcFrame;      // push dword ptr fs:[eax]
    desc: PExcDesc;       // push $0044ca0b
    hEBP: Pointer;        // push ebp
    case Integer of
    0:  ( );              // for normal finally
    1:  ( ConstructedObject: Pointer );
    2:  ( SelfOfMethod: Pointer );
  end;
```

操作系统约定：FS:[0]中总是存放当前(最内层)异常帧的首地址。因此，与使用“push dword ptr fs:[eax]”语句来取出 *TExcFrame.next* 相对应，使用“mov fs:[eax],esp”语句，也就是向异常帧链表中插入最新的帧。

在插入新的异常帧后，编译器实现语句：

```
Unit1.pas.37: R.S := '';
0044C9E7 8D45FC          lea eax,[ebp-$04]
0044C9EA E882174FBFF     call @LStrClr
```

Delphi 源代码分析

接下来，编译器加入结束化处理部分的代码。首先要从栈上清除异常帧，并将 FS:[0]恢复为上一个异常帧的指针：

```
0044C9EF 33C0          xor eax,eax
0044C9F1 5A            pop edx                 // 弹出上一个异常帧地址：TExcFrame.next
0044C9F2 59            pop ecx                 // 弹出TExcFrame.desc
0044C9F3 59            pop ecx                 // 弹出TExcFrame.hEBP
0044C9F4 648910        mov fs:[eax],edx        // 将上一个异常帧地址送入FS:[0]，恢复。
```

从$0044CA0B 开始的 7 个字节，是一个异常描述记录 TExcDesc，它的指针被填入 TExcFrame.desc 域中。其结构定义如下：

```
type
 JmpInstruction =
 packed record
   opCode:   Byte;
   distance: Longint;
 end;

 PExcDesc = ^TExcDesc;
 TExcDesc =
 packed record
   jmp: JmpInstruction;
   case Integer of
   0:      (instructions: array [0..0] of Byte);
   // ...
 end;
```

如果将本例中的 7 个字节，映射到 ExcDesc 记录中，其各域的值为：

```
ExcDesc.jmp.opCode          : $E9          // JMP
ExcDesc.jmp.distance        : $046EFBFF    // @HandleFinally
ExcDesc.instructions[0]     : $EB          // JMP
ExcDesc.instructions[1]     : $EA          // -$16
```

按照操作系统的约定，一旦发生异常，操作系统将依据 FS:[0]的值，调用 *ExcDesc.jmp* 记录的首地址，也就是说开始执行如下的代码：

```
0044CA0B E9046EFBFF     jmp @HandleFinally
0044CA10 EBEA           jmp -$16
```

调用 Delphi 的异常处理代码(_HandleFinally 例程)的这一部分的代码藏在 *ntdll.dll* 的导出例程 KiUserExceptionDispatcher()中：

```
ntdll.KiUserExceptionDispatcher:
```

```
77F35267 8B4C2404          mov ecx,[esp+$04]
77F3526B 8B1C24            mov ebx,[esp]
// ...
77F58CD0 55                push ebp
77F58CD1 8BEC              mov ebp,esp      // 当前栈顶，[esp+4]为Delphi中的异常帧
77F58CD3 FF750C            push dword ptr [ebp+$0c]
77F58CD6 52                push edx         // 压入一个异常记录，OS需要保护下面的代码
77F58CD7 64FF3500000000    push dword ptr fs:[$00000000]
77F58CDE 64892500000000    mov fs:[$00000000],esp

//操作堆栈，为HandleFinally()传参数
77F58CE5 FF7514            push dword ptr [ebp+$14]
77F58CE8 FF7510            push dword ptr [ebp+$10]
77F58CEB FF750C            push dword ptr [ebp+$0c]
77F58CEE FF7508            push dword ptr [ebp+$08]
77F58CF1 8B4D18            mov ecx,[ebp+$18]
77F58CF4 FFD1              call ecx         // CALL TExcFrame.desc.jmp
```

从这里可以看出，操作系统并不知道，也根本不认为在TExcFrame.desc.jmp后面还有TExcFrame.desc.instructions结构的存在。事实上，instructions结构是归由Delphi内部例程_HandleFinally()来使用的。因此，一旦操作系统将控制权交还给_HandleFinally()，就不再关心其具体的行为了。

所以，在最小化内核中，只须加入一个例程_HandleFinally()就可以“支持”异常处理了——在OS的异常帧链表加入和撤消_HandleFinally()的代码是由编译器负责的。当然，使用最小化内核时应当清楚：最小化内核的_HandleFinally()其实什么也不做。

8.4.2　从操作系统的角度来理解编译器行为

FS:[0]中存放一个异常帧的指针，是操作系统的约定。这个约定的细节是这样的：当线程初始化准备好运行时FS段寄存器指向一个TIB结构(Thread Information Block)。而这个结构定义如下：

```
PEXCEPTION_REGISTRATION_RECORD = ^EXCEPTION_REGISTRATION_RECORD;
_EXCEPTION_REGISTRATION_RECORD = packed record
  pNext: PEXCEPTION_REGISTRATION_RECORD;
  pfnHandler: FARPROC;
end;
EXCEPTION_REGISTRATION_RECORD = _EXCEPTION_REGISTRATION_RECORD;
TExceptionRegistrationRecord = EXCEPTION_REGISTRATION_RECORD;
PExceptionRegistrationRecord = PEXCEPTION_REGISTRATION_RECORD;

_NT_TIB = record
```

```
    // 00h Head of exception record list
    ExceptionList: PEXCEPTION_REGISTRATION_RECORD;
    StackBase: PVOID;          // 04h Top of user stack
    StackLimit: PVOID;         // 08h Base of user stack
    // ...
end;
```

可见，FS:[0]正好是"异常注册记录链表"的表首指针 ExceptionList(类似地，FS:[4]就是栈的基底地址 StackBase)。

在约定中，操作系统仅知道 TIB 中的 ExceptionList 所指向数据的基本结构(TExceptionRegistrationRecord)，甚至不完全清楚这些数据的具体细节。它也并没有约定以何种方式来形成异常记录链表。因此，应用程序可以在数据区、代码区、栈、堆(Heap)等内存结构中存放链表结点，只要保证操作系统可以执行到表首结点的 pfnHandler 域即可。——在应用程序发生异常时，操作系统将把控制权移交给 pfnHandler 域所指向的代码。

因此，操作系统本身并不知道"异常处理"和"结束处理"的存在，它只知道 ExceptionList 的存在。而对于 Delphi 来说，ExceptionList 被实现为"异常帧(TExcFrame)链表"。

除了称谓上的不同，"异常帧"还体现了 Delphi 对 ExceptionList 的具体实现方式：在栈上存储异常注册记录，并将记录结构进一步地扩展成为 TExcFrame。之所以 Delphi 需要这样做，是因为它必须要同时实现"异常处理"和"结束处理"两种 SEH 的行为。

操作系统一旦发生异常，就调用 ntdll.KiUserExceptionDispatcher()，所以在 OS 一级，可以认为一切都从 KiUserExceptionDispatcher()开始[①]：

```
procedure KiUserExceptionDispatcher;
var
  Proc : procedure (PExceptionRecord, PExceptionRegistration, PContext,
    PParam); stdcall;
  PFrame : ^TExcFrame;
  PExcRec : ^TExceptionRecord;
begin
  // ...
  try
    { // call +$00000000
      // ...
      // push ebp
```

① 以下的代码描述都是形式化的。

```
    // mov ebp, esp
    -> [EBP]        = save EBP by this call
       [EBP+  4]    = Caller
       [EBP+  8]    = save by KiUserExceptionDispatcher(), TExceptionRecord
       [EBP+ $C]    = save Delphi's ExcFrame in Stack
       [EBP+$10]    = ctxPtr, PContext
       [EBP+$14]    = dspPtr, PParam
       [EBP+$18]    = save Delphi's ExtDesc
  }
  PExcRec := GetExceptionRecordFromStack;      // [ESP + $08]
  PFrame := GetFrameFromStack;                 // [ESP + $0C]
  Proc := GetHandleProcFromStack;              // [ESP + $18]
  Proc(PExcRec, PFrame, ctxPtr, dspPtr);
 finally
  // ...
 end;
end;
```

接下来，并不会马上执行到 Delphi 中的_HandleFinally()，因为 TExcFrame.desc 并不指向_HandleFinally()，而是指向如下的这样一个例程：

```
procedure HandleInHere;
asm
       { ->     [ESP+ 4] excPtr: PExceptionRecord      }
       {        [ESP+ 8] errPtr: PExcFrame             }
       {        [ESP+12] ctxPtr: Pointer               }
       {        [ESP+16] dspPtr: Pointer               }
       { <-     EAX return value - always one          }
  jmp @HandleFinally
end;
```

_HandleFinally()做些什么呢？以下四行代码已经决定了一切：

```
procedure      _HandleFinally;
// ...
  MOV       EDX,[ESP+8]                    // [ESP+ 8] errPtr: PExcFrame
  MOV       ECX,[EDX].TExcFrame.desc       // 定位到 ExcFrame.desc.jmp
  ADD       ECX,TExcDesc.instructions      // 定位到 ExcFrame.desc.instructions
  CALL      ECX
// ...
```

因此，真正的 finally 保护的代码在这里：

```
procedure FinallyInHere;
asm
```

Delphi 源代码分析

```
  // ExcFrame.desc.instructions
  jmp -$16
end;
```

"-$16"是编译期计算出来的，instructions也是在编译期填写好的一个结构(数组)。SEH只是给开发者一个组织这样的结构的机会。Delphi的实现方式，简单地说，是让编译器填写好instructions，然后OS和ExcFrame.desc.jmp最终总是会执行到instructions所指向的代码。

需要留意的是，在调用FinallyInHere()时，寄存器EBP是被恢复的。只有这样，才能保证执行finally-end中的代码时，能够准确地访问到各个入口参数和局部变量。恢复EBP的行为也是由_HandleFinally()来实现的。而做到这一点也并不难，因为ExcFrame.hEBP中已经保存了最原始(进入异常处理之前)的EBP值。

8.4.3 try..findlly/except..end 语法关键字与内部例程

Delphi本身只关注两件事：

- 如何将异常处理例程挂接到OS的FS:[0]上，以及何时清除；
- 当OS发现异常时，如何在处理例程中响应异常。

关于第一件事，除非必须关注于SEH的实现机制，绝大多数时候，开发人员是不必考虑操作系统"何时"以及"怎样"使执行流程进入到异常处理例程中去的。这是操作系统与编译器之前交互的结果。

而对于第二件事，在Delphi内部的异常处理例程(在*System.pas*中)只有如下四个：

```
// 响应 except .. end
procedure _HandleAnyException;

// 响应 finally .. end
procedure _HandleFinally;

// 响应 except .. (on ... do) .. end
procedure _HandleOnException;

// 响应自动化控制过程(automated)中的异常
procedure _HandleAutoException;
```

因此，try-finally/except-end更多的时候仅被当成关键字来看待，而忽略编译器对它们的具体实现。如果需要了解这些关键字与内部例程之间是如何协作的，表8-1将是一个很

好的开端。

表 8-1　关键字与内部例程的协作关系

关键字	代　码	作　用
try	xor eax,eax push ebp push @@HandleInHere push dword ptr fs:[eax] mov fs:[eax],esp	(在栈上)形成异常帧，并插入到链表(FS:[0])头部
finally ... end	// ... // do user code xor eax,eax pop edx pop ecx pop ecx mov fs:[eax],edx push @@OK @@FinallyCode: // ... // finally notify code // return to @@OK ret @@HandleInHere: jmp @HandleFinally jmp @@FinallyCode @@OK // ...	保证finally notify code总是能被执行到。 从链表中清除被响应的异常帧
except ... end	// ... // do user code xor eax,eax pop edx pop ecx pop ecx mov fs:[eax],edx jmp @@OK @@HandleInHere: jmp @HandleAnyException // ... // except notify code call @DoneExcept @@OK: // ...	保证异常发生时，except notify code会被执行。 从链表中清除被响应的异常帧

8.5 基本(except 型)异常处理

大多数时候，开发人员会用如下的结构来处理异常：

```
try
  // user code
except
  // except notify code
end;
```

Delphi 在内核中实现了如下例程以支持这种结构(参见上一小节)：

```
// 异常处理句柄(部分)
procedure _HandleAnyException;

// 异常的结束化处理
procedure _DoneExcept;
```

编译器保证用户的“except notify code”被包含在_HandleAnyException()和_DoneExcept()例程之间；而操作系统的 SEH 机制将保证在“user code”引发异常时，执行流程总是进入到_HandleAnyException()或者其他的一些处理句柄。

如果所有的“except notify code”都不处理异常，则将导致异常被送至最外层的处理程序_ExceptionHandler()——除此之外，异常总能在一个 except-end 块中被处理完毕，执行流程也总是能在例程_DoneExcept()结束后，进入其后的代码。

然而一切并非像表面这样。首先应该清楚地知道的是：在 *System.pas* 中，Delphi 仅仅实现了异常处理的框架，以及部分代码。

Delphi 需要向开发者提供的是一个完整的“面向对象的异常处理构架”，而 *System.pas* 只实现了基本的运行期库(RTL)支持，并不包含更多的类(Class)。因此，异常处理(try-except-end)真正的代码实现部分，被放在了 *SysUtils.pas* 中[①]。

① 本小节只讨论 except 型异常处理在 *System.pas* 单元中的实现结构。except 型异常处理的具体实现是由 *SysUtils.pas* 中的 *Exception* 类来完成的，这些将留在下一节的“面向对象的异常处理构架”中详细讲述。

8.5.1　异常触发(Raise)

除了代码执行过程中因为未知的情况引发异常之外，开发者还可以使用 raise 关键字来手工引发异常。raise 关键字通过以下两个例程来实现：

```
// 引发指定异常
procedure _RaiseExcept;

// 重新引发当前正在处理的异常
procedure _RaiseAgain;
```

raise 关键字的细节涉及到异常记录(TExceptionRecord)和异常类(Exception Classes)的实现。接下来的一些小节中将会对此详细叙述，这些内容分布在：

- 异常记录(TExceptionRecord)与异常触发(RaiseException)的实现："顶层异常处理"小节；
- raise 关键字的实现：第 6 节的"_RaiseExcept()、_RaiseAgain()与_DoneExcept()例程"小节；
- 异常类型的识别：第6节的"异常对象"和"异常类型检测的依赖性问题"小节。

8.5.2　多层(嵌套)的异常处理

异常处理例程是操作系统调用的一个子例程，其出口和入口时的参数表类似于：

```
procedure      _HandleAnyException;
asm
        { ->    [ESP+ 4] excPtr: PExceptionRecord      }
        {       [ESP+ 8] errPtr: PExtFrame             }
        {       [ESP+12] ctxPtr: Pointer               }
        {       [ESP+16] dspPtr: Pointer               }
        { <-    EAX return value - always one          }
// ...
```

excPtr 中是操作系统约定的一个数据结构，由 KiUserExceptionDispatcher()传给用户的异常处理例程：

```
// define in windows.pas
  _EXCEPTION_RECORD = record
    ExceptionCode: DWORD;                    // 异常代码
    ExceptionFlags: DWORD;                   // 异常标志
    ExceptionRecord: PExceptionRecord;       // 指向另外一个EXCEPTION_RECORD的指针
    ExceptionAddress: Pointer;               // 异常发生的地址
```

```
  NumberParameters: DWORD;                    // ExceptionInformation数组的元素数
  ExceptionInformation: array[0..EXCEPTION_MAXIMUM_PARAMETERS - 1]
    of DWORD;
end;
```

Delphi 将_EXCEPTION_RECORD 描述为 ExceptionInformation 可变的记录：

```
PExceptionRecord = ^TExceptionRecord;
TExceptionRecord =
  // ...
  case (IsDelException:) Boolean of
  True:  (ExceptionInformation : array [0..14] of Longint);
  False: (ExceptAddr: Pointer; ExceptObject: Pointer);
end;
```

在这个记录中，ExceptionFlags 字段表明异常的特征和它所处于的状态：

```
cContinuable        = 0;      // 可修复异常
cNonContinuable     = 1;      // 不可修复异常
cUnwinding          = 2;      // 正在展开，不要试图修复什么，需要的话，释放必要的资源
cUnwindingForExit   = 4;      // 退出
cUnwindInProgress = cUnwinding or cUnwindingForExit;
```

_HandleAnyException() 检测 ExceptionFlags 的值，如果是 cUnwinding 或 cUnwindingForExit，那么将直接退出。否则将试图处理异常和执行 except-end 中的代码。

异常处理程序的返回值可以是如下之一：

```
EXCEPTION_EXECUTE_HANDLER       = 1;    // 继续执行(上一层的)异常处理例程
EXCEPTION_CONTINUE_SEARCH       = 0;    // 继续处理，通常
EXCEPTION_CONTINUE_EXECUTION    = -1;   // 从异常位置尝试重新执行
```

无论是否处理异常，_HandleAnyException() 出口值总是为 1，这使得 KiUserExceptionDispatcher 将把异常重新提交给上一层的异常处理例程。

Delphi 没有指定 EXCEPTION_CONTINUE_EXECUTION 返回参数的情况，如果用户认为有必要那样做，那么应该重写缺省异常处理程序，并且在返回 EXCEPTION_CONTINUE_EXECUTION 之前，修改异常记录的上下文，使得代码可正常执行，而不会重新引发异常。

上面这些内容，只是说明：如果一个异常不被内层的处理程序处理，那么它将被 KiUserExceptionDispatcher()提交给上一层的处理程序，直到 FS[0]指向的链表的末尾结点。在末尾结点上，至少会有一个“顶层异常处理程序”，那通常是操作系统为每个线程缺省准备的。

8.5.3　异常展开(Unwind)

异常“被交给上一层处理程序”并不是“异常展开”。异常展开是用户代码的行为，而不是操作系统的行为。也就是说，“异常展开”是由 *System.pas* 中的代码来完成的，而不是在KiUserExceptionDispatcher()中自动完成的。

异常展开是异常处理例程在决定处理某个异常的时候，给前面不处理这个异常的处理例程的一个清洗的机会。也就是说，当异常被处理(不包括提交到“上一层处理程序”)时，可以给内层(所有)的处理程序一个信号，由它们自行决定是否进行一些最后的操作。

对内层处理程序来说，“最后的处理”真的很重要吗？居然值得外层处理程序为此专门进行“异常展开”。

以下面的代码为例：

```
var
  AFileName : String;

procedure TestUnwindChild;
var
  F : THandle;
begin
  F := FileOpen(AFileName, fmOpenWrite);

  try
    raise TException.Create('一个未知的异常.');
  except
    on E: EHeapException do
      writeln('一些已知的异常得到处理');
  end;

  FileClose(F);
end;

procedure TestUnwindMain;
var
  F : THandle;
begin
  try
    TestUnwindChild;
  except
    writeln('处理任意的、未知的异常.');
  end;
```

```
  // 现在，如果TestUnwindChild()中出现未知异常，AFileName所指定的文件将无法打开了
  F := FileOpen(AFileName, fmOpenWrite);

  // ...
end;
```

这样的代码的确显得拙劣，但“绝对优质”的代码根本就用不着异常处理(当然，这样的代码也不存在)。所以异常展开机制的存在，就是为内层代码留下一个处理共享对象、锁、句柄或者堆栈等等的机会。在上面的例子中，TestUnwindChild()就需要得到一个关闭文件句柄 F 的机会。

很显然，如果要留下这样一个机会，通常的做法是用 try-finally-end 块保护代码。而 Delphi 中的 Unwind 也正是为 finally 异常而保留的。

System.pas 中，_HandleFinally()是惟一一个响应“展开的异常”的例程：

```
procedure       _HandleFinally;
// ...
        TEST    [EAX].TExceptionRecord.ExceptionFlags,cUnwindInProgress
        JNE     @@exit

        // ...
        // Finally notify code
@@exit:
        XOR     EAX,EAX
end;
```

除此之外，所有四个响应异常的例程所使用的代码都是一致的：对 Unwind 不进行任何处理。下面的代码说明这一点：

```
// procedure        _ExceptionHandler;
// procedure        _HandleAutoException;
// procedure        _HandleOnException;
procedure           _HandleAnyException;
// ...
        MOV     EAX,[ESP+4]
        TEST    [EAX].TExceptionRecord.ExceptionFlags,cUnwindInProgress
        JNE     @@exit
// ...
```

因此，一个异常发生时：

- 内层的 finally 型处理将先忽略异常；

- 外层的 except 型处理接到异常，在进行处理之前，先向内层展开异常；
- 内层的 finally notify code 接到展开的异常，处理，返回；
- 外层的 except notify code 处理异常。

可见，一个 finally 型处理总是被保护于一个外层的 except 型处理之中。这样，假使内层的 finally 处理过程中，又有新的异常引发，那么由于新异常发生在 except-end 保护块中，因此新异常(在 except-end 块中)的展开并不会导致内层的 finally-end 参与处理——如果它参与处理，那么又会再次产生异常，于是就死循环了。

异常处理例程调用 RtlUnwindProc 所指向的例程来实现异常展开，缺省时该系统全局变量指向 API：RtlUnwind()。

```
// define in WinLoru1.h, in MSDN
procedure RtlUnwind(TargetFrame: Pointer; TargetIp: Pointer;
  ExceptionRecord:PExceptionRecord; ReturnValue: Pointer); stdcall;
```

一段典型的调用 RtlUnwind()的代码是这样的：

```
// code from System.pas, _HandleAnyException()
@@GoUnwind:
// -> [ESP+ 8] errPtr; PExcFrame
// ...
        PUSH    EBX             // 暂存 EBX
        XOR     EBX,EBX         // EBX置0
        PUSH    ESI             // 暂存三个重要的寄存器(必须)
        PUSH    EDI
        PUSH    EBP

        MOV     EBX,FS:[EBX]            // 取异常帧链表的表首FS:[0]
        PUSH    EBX                     // Save pointer to topmost frame
        PUSH    EAX                     // Save OS exception pointer
        PUSH    EDX                     // Save exception object
        PUSH    ECX                     // Save exception address

        // 计算当前异常帧在栈上的位置(前面压入了8个32位值)
        MOV     EDX,[ESP+8+8*4]

        PUSH    0                               // ReturnValue = 0
        PUSH    EAX                             // ExceptionRecord = 当前的异常记录
        PUSH    offset @@ReturnAddress          // TargetIp = 返回时的地址
        PUSH    EDX                             // TargetFrame = 当前异常帧
        CALL    RtlUnwindProc
// ...
```

在调用前暂存数据的不同，是上述的四个异常处理例程在调用 RtlUnwind()时的主要区别。此外，在进入@@GoUnwind 之前的入栈值的数量，也会影响 EDX 的计算方法。例如：

```
procedure       _HandleOnException;
// ...
@@GoUnwind:
        XOR     EBX,EBX
        MOV     EBX,FS:[EBX]
        PUSH    EBX                     { Save topmost frame      }
        PUSH    EAX                     { Save exception record   }
        PUSH    EDX                     { Save exception object   }
        PUSH    ECX                     { Save exception address  }

        MOV     EDX,[ESP+8+8*4]
// ...
```

这里在进入@@GoUnwind 之后只入栈了四个值，但计算时却用了 8 个 32 位值(8*4)，这是因为在@@common 标签后压了四个值入栈：

```
procedure       _HandleOnException;
// ...
@@common:

        MOV     EDX,[ESP-8]

        PUSH    EBX
        PUSH    ESI
        PUSH    EDI
        PUSH    EBP
```

8.5.4 异常响应(Notify)

根据用户代码中具体处理方法的不同，异常响应中可能发生的情况是如下三种之一：

- 不进行筛选，对所有异常使用同一种方法处理。这是 except-end 型的异常处理。
- 进行筛选，符合条件的处理，其他的使之送至外层处理程序。这是 except-on..do-end 型的异常处理。

此外，对操作系统来说，finally 型中的代码也是对异常做出的响应。Delphi 据此实现了三个不同的异常处理(响应)句柄：

```
// except-on..do-end型异常处理句柄
procedure _HandleOnException;
```

```
// except-end型异常处理句柄
procedure _HandleAnyException;

// finally-end型异常处理句柄
procedure _HandleFinally;
```

每个响应异常的句柄通常包括两种行为：

- 由用户的异常响应代码来进行处理(包括重发)。
- 内核需要得到一个执行权限，来给 IDE 内置调试器准备信息。

Delphi 用两行很简单的代码来实现这两种行为。其中的第一行是调用 Delphi 内核的异常响应例程，包括：

```
// 由_HandleAnyException()调用
procedure       NotifyAnyExcept;

// 由_HandleOnException()调用
procedure       NotifyOnExcept;

// 由_HandleFinally()调用
procedure       NotifyExceptFinally;
```

这些例程将检测系统全局变量 **DebugHook** 的状态，以决定是否为 IDE 内置调试器准备信息。

第二行则是用于进入用户的异常响应代码。这两行代码如下：

```
// ECX 指向 TExcDesc.instructions
procedure       _HandleFinally;
// ...
        CALL    NotifyExceptFinally
        CALL    ECX
end;

// EBX 指向一个异常描述入口结构 TExcDescEntry
procedure       _HandleOnException;
// ...
        CALL    NotifyOnExcept
        JMP     [EBX].TExcDescEntry.handler
end;

// EBX 指向 TExcDesc.instructions
procedure       _HandleAnyException;
```

Delphi 源代码分析

```
// ...
        CALL    NotifyAnyExcept
        JMP     EBX
end;
```

由于 finally 型只处理展开的异常，处理完后需要返回到上一层处理的 RtlUnwindProc() 调用之后执行，因此它使用“CALL”调用用户的处理代码；而 except 型在执行完用户代码后，总是会调用_DoneExcept()来结束异常处理，并返回到 except-end 结构后继续执行，所以它使用“JMP”来进入用户的处理代码—— 一旦进入 except 的用户异常响应代码，程序的执行流程就不会再返回到例程 procedure _HandleXXXX()。

异常的筛选是在异常展开和响应之前就进行的。Delphi 使用例程 FindOnExceptionDescEntry()，基于对象信息来实现筛选。其细节参见本章第 6 节“使用面向对象技术的异常类”。

8.5.5 顶层异常处理

按照操作系统的约定，可以使用 API：SetUnhandledExceptionFilter()来设定一个顶层异常处理句柄，以处理所有在用户代码中未处理的异常。

但是 Delphi 内核中却并没有用这样的方式来实现一个顶层异常处理。在.*EXE* 和.*DLL* 两种类型的模块的初始化代码中，Delphi 加入一个顶层的异常处理过程，这样就可以捕获到从用户代码中逃逸的所有异常。[①]

这个顶层异常处理过程就是_ExceptionHandler()：

```
// 顶层异常处理例程
procedure       _ExceptionHandler;

// 在当前模块中加入顶层异常处理例程
procedure       SetExceptionHandler;

// 在当前模块中清除顶层异常处理例程
procedure       UnsetExceptionHandler;
```

不使用 API：SetUnhandledExceptionFilter()，而是在_StartExe() 和

① 初始化代码中如何加入这个顶层异常处理过程的细节，可以参见第 5 章“其他”小节中的“入口代码中的堆栈使用深入分析(内存现场)”。

_StartDll()中增加调用 SetExceptionHandler()来设置顶层异常处理句柄的原因，是与异常识别、模块识别、异常展开以及多线程相关的。

通常一个模块并不是独立运行的。同一进程中的模块可能是用 C、Delphi、汇编或者其他任意一种未知的语言编写的，因此不同模块(的编译器)对 SEH 具体的实现并不相同。而操作系统只约定了异常帧的基本结构，所以各模块之间除了能使用相同的方法遍历异常帧链表之外，根本没有办法去进行其他形式的协同处理。

对于操作系统来说，只约定了 FS:[0]存放当前线程的异常帧链表表首，因此任何的模块都必须使用这个地址来存取异常帧。而 SetUnhandledExceptionFilter()是针对所有线程的，一个"UnhandledExceptionFilter"用于处理所有线程的所有未被响应的(Unhandled)异常。因此如果 Delphi 想在这里设置自己的"顶层异常处理"例程，那么必将对所有线程进行辨识处理。显然，辨识线程并不会比在创建线程时加入一段代码来得容易[1]。

同样的道理，Delphi 在模块的入口代码处加入"顶层异常处理"，也使得模块的识别相对简易。不过更加直接的原因是：Delphi 希望用_ExceptionHandler()来响应所有由 Delphi 代码触发的异常，而将其他未知类型的异常交给 UnhandledExceptionFilter，同时尽可能少地在_ExceptionHandler()中进行异常来源的辨识。

_ExceptionHandler()首先无条件地向内层的异常处理例程展开异常，这使得 try-finally-end 总是能得到处理。接下来，_ExceptionHandler()对它捕获到的异常记录进行辨识。如果它是一个 Delphi 异常，则调用 NotifyUnhandled()来通知调试器响应该异常(仅对处于调试状态时有意义)，然后通过检测系统全局变量 ExceptProc，来判定异常类是否已经包含在当前模块中。仅有当 *SysUtils.pas* 的单元初始化完成，且 ExceptProc 被设置时，该异常才会得到响应。

为了知道哪个异常是 Delphi 的代码发出的，Delphi 在代码中实现这样的约定：在 *System.pas* 和 *SysInit.pas* 之外的单元中，都使用 raise 关键字发出异常。

Delphi 能够识别三类异常，并分别处理：

- 对于由 raise 关键字发出的异常，将作为 Exception 或其子类对象处理；
- 对于 C++ Builder 模块发出的异常，将强制为 Delphi 的异常对象处理；
- 如果异常不是由 Delphi 发出的，而是操作系统异常，那么这个异常将通过 MapToRunError()映射到内核的"错误"处理系统中；如果 *SysUtils.pas* 单元被引用，那么该异常将通过 MapException()例程映射到一个具体的 Delphi 异常类。

[1] Delphi 为每个自己创建的线程加入一个"顶层异常处理"，在多线程的实现机制中可以看到相关的代码。

为此，Borland 在 Delphi 和 C++ Builder 之间做出约定，在异常记录 TExceptionRecord 的 ExceptionCode 中存放编译器识别代码：

```
cDelphiException    = $0EEDFADE;
cDelphiReRaise      = $0EEDFADF;
cDelphiExcept       = $0EEDFAE0;
cDelphiFinally      = $0EEDFAE1;
cDelphiTerminate    = $0EEDFAE2;
cDelphiUnhandled    = $0EEDFAE3;
cNonDelphiException = $0EEDFAE4;
cDelphiExitFinally  = $0EEDFAE5;
cCppException       = $0EEFFACE; { used by BCB }
```

而 raise 关键字的基本实现代码如下：

```
procedure       _RaiseExcept;
asm
// ...
    PUSH    ESP                 { pass pointer to arguments       }
    PUSH    7                   { there are seven arguments       }
    PUSH    cNonContinuable     { we can't continue execution     }
    PUSH    cDelphiException    { our magic exception code         }
    PUSH    EDX                 { pass the user's return address  }
    JMP     RaiseExceptionProc
end;
```

这用于在栈上构成一个 TExceptionRecord 记录：

```
PExceptionRecord = ^TExceptionRecord;
TExceptionRecord =
record
  ExceptionCode         : LongWord;
  ExceptionFlags        : LongWord;
  OuterException        : PExceptionRecord;
  ExceptionAddress      : Pointer;
  NumberParameters      : Longint;
  case (IsOsException:) Boolean of
    True:  (ExceptionInformation : array [0..14] of Longint);
    False: (ExceptAddr: Pointer; ExceptObject: Pointer);
end;
```

这样，每一个通过 raise 关键字发出的异常的 ExceptionFlags 域都会是 cDelphiException，而一旦 RaiseExceptionProc 通过调用 API：RaiseException() 发出异常，那么 TExceptionRecord 与 TExcFrame 将被作为参数传到异常响应句柄。

因此，在顶层异常处理例程中可以用如下的代码来识别这些异常：

```
procedure       _HandleAnyException;
// ...
@@ExecuteHandler:
        // ...
        PUSH      0
        PUSH      EAX
        PUSH      offset @@returnAddress
        PUSH      EDX
        CALL      RtlUnwindProc     // 进行异常展开，展开后返回到 @@returnAddress

@@returnAddress:
        MOV       EBX,[ESP+4]
        CMP       [EBX].TExceptionRecord.ExceptionCode,cDelphiException
        MOV       EDX,[EBX].TExceptionRecord.ExceptAddr
        MOV       EAX,[EBX].TExceptionRecord.ExceptObject

      //如果是Delphi中的异常类，则进入@@DelphiException2
      JE          @@DelphiException2
// ...
@@DelphiException2:
// ...
```

在其他的异常响应句柄中，对 cDelphiException 的识别也是与此类同的。

8.6　使用面向对象技术的异常类

在 *System.pas* 中，Delphi 完整地实现了“结束处理(finally)”，但是仅实现了“异常处理(except)”的框架。它向外部代码提供了一系列的全局变量和例程，用以实现真正的异常处理机制。这包括：

```
var
  ExceptProc: Pointer;        // 所有未被响应的(Unhandled)异常的响应例程①
```

① 在 ExceptionHandler()中使用该过程指针，可以指定一个全局的未响应异常处理程序。在处理程序中，可以响应的异常不但包括在 try-except 块中引发(且未被处理)的异常，也包括一些操作系统的异常——《Delphi 技术手册》在这一点的描述上是不完整的。

```
  ExceptClsProc: Pointer;   // 在except-on..do-end型异常处理中使用
  ExceptObjProc: Pointer;   // 将一个操作系统异常映射到Delphi的异常类实例上使用

  // 引发异常的例程。通常指向API: RaiseException()
  RaiseExceptionProc: Pointer;
  RTLUnwindProc: Pointer;   // 展开异常的例程。通常指向API: RTLUnwind()
  ExceptionClass: TClass;   // IDE调试器响应的异常基类。

// 异常对象的申请和释放
function AcquireExceptionObject: Pointer;
procedure ReleaseExceptionObject;

// 当前异常列表
function RaiseList: Pointer; deprecated;
// 加入一个异常到列表表首，返回上一个表首值
function SetRaiseList(NewPtr: Pointer): Pointer; deprecated;

// 当前异常对象和异常地址信息
function ExceptObject: TObject;
function ExceptAddr: Pointer;
```

8.6.1 异常列表、RaiseFrame 与 ExceptionRecord

System.pas 会维护一个异常列表 RaiseList。这个列表被描述为“当前异常对象栈（Stack of current exception objects）”。

这里的“栈”是指这些异常对象是以“栈”这种数据结构保存的，而并不明确指出异常对象在栈上分配。准确地说，RaiseList 保存的是异常对象的“触发记录”，而非异常对象本身。

异常的“触发记录”的类型为 TRaiseFrame：

```
PRaiseFrame = ^TRaiseFrame;
TRaiseFrame = packed record
  NextRaise: PRaiseFrame;
  ExceptAddr: Pointer;
  ExceptObject: TObject;
  ExceptionRecord: PExceptionRecord;
end;
```

触发记录也在栈上分配，其分配栈的代码开始于异常展开之前：

```
procedure       _HandleAnyException;
// ...
      MOV       EBX,FS:[EBX]
```

```
        PUSH    EBX             // 暂存异常链表的表首FS:[0]，即: topmost frame
        PUSH    EAX             // 入栈: TRaiseFrame.ExceptionRecord
        PUSH    EDX             // 入栈: TRaiseFrame.ExceptObject
        PUSH    ECX             // 入栈: TRaiseFrame.ExceptAddr

        // ...
        PUSH    offset @@ReturnAddress
        PUSH    EDX
        CALL    @RtlUnwindProc          // 异常展开
@@ReturnAddress:
        // ...

        {       Make the RaiseList entry on the stack }
        CALL    SysInit.@GetTLS
        PUSH    [EAX].RaiseListPtr      // 入栈: TRaiseFrame.NextRaise
```

异常的触发记录是在异常展开结束后才添加到 RaiseList 链表中的，在异常展开的时候，在 RaiseList 上并没有当前异常的触发记录(指针)。因此并不能在 finally 型处理中知道是否有异常发生。亦即是说，无法判断是因为异常而进入 finally-end 块中，还是在 try-finally 块正常执行结束后进入了 finally-end 块。[①]

触发记录(RaiseFrame)是由异常响应句柄在栈上分配的，而异常记录(ExceptionRecord)是由异常引发例程_RaiseExcept()在栈上分配的。那么，如果一个异常不是由 raise 触发的，它会有触发记录吗？

这一点与 RaiseList 这个变量名称的表面含义并不相同。一个异常即使不是由 raise 引发的，而是由操作系统或者其他模块的导入例程引发的，它仍然会在异常响应句柄捕获到它的时候，在栈上形成一个 TRaiseFrame。惟一的条件是：异常响应句柄能够识别该异常[②]。

8.6.2　未知异常映射：ExceptObjProc

显然，如果要在栈上形成一个 TRaiseFrame 记录，则必然要填写它的 ExceptObject 和 ExceptionRecord 域。然而如果一个异常是由操作系统或者其他模块的导入例程引发的，那么就没有 ExceptObject 和 Delphi 扩展类型的 ExceptionRecord。

① 的确不可能通过 RaiseList 来检测这时是否有异常发生，但是可以通过检测堆栈现场来判定。Beta 和我关于这个问题的讨论全文发布在大富翁论坛上：http://www.delphibbs.com/delphibbs/dispq.asp?lid=2211454

② 参见 8.5.5 小节“顶层异常处理”。

因此，Delphi 使用异常转换的方式，处理两种类型的异常：

- 使用 ExceptObjProc 来将“操作系统异常”[①]映射为 Delphi 异常对象。
- 如果是 C++ Builder 的异常，则强制为 Delphi 异常处理。

在 *SysUtils.pas* 中，ExceptObjProc 被赋值为指向 GetExceptionObject()的指针。该例程通过 MapException()将一个未知异常映射为 Delphi 运行期错误(TRuntimeError)，并取其序号作为错误代码(ErrorCode)，最后根据错误代码来查找相应的异常类并创建异常对象。

在异常响应句柄中，如果未知异常能被影射到异常对象(调用 ExceptObjProc 时返回值不是 nil)，则检测它是不是 C++ Builder 代码引发的异常：

```
procedure       _HandleAnyException;
// ...
    MOV     EDX,ExceptObjProc
    TEST    EDX,EDX         // ExceptObjProc是否置值
    JE      @@exit
    CALL    EDX             // Call ExceptObjProc()
    TEST    EAX,EAX         // 返回值非nil，则影射成功
    JE      @@exit
    MOV     EDX,[ESP+12]
    MOV     ECX,[ESP+4]
    CMP     [ECX].TExceptionRecord.ExceptionCode,cCppException //检查识别码
    JE      @@CppException
    // ...

@@CppException:
    MOV     EDX,EAX             // 将通过ExceptObjProc得到的异常对象送入 EDX
    MOV     EAX,[ESP+4]         // 将 PExceptionRecord 指针送入 EAX
    MOV     ECX,[EAX].TExceptionRecord.ExceptionAddress  //异常地址送入ECX

@@DelphiException:
    CMP     BYTE PTR JITEnable,1
// ...
```

这里在“@@CppException:”标签处给 EDX、EAX、ECX 寄存器传值，就是为了在后面维护 RaiseList 时，向堆栈压入一个 TRaiseFrame 记录而使用的。此后，栈状态、寄存器状态与引发 Delphi 异常时是一致的。这其实也是将 C++ Builder 异常强制为 Delphi 异常的代码。

① 在《Delphi 技术手册》中，它被称为“Windows 异常”，含义是相同的。

接下来，如果未知异常并非 C++ Builder 发出的，则先调用 NotifyNonDelphiException()来通知调试器(如果它存在的话)有一个未知类型的异常出现。接下来异常会被发送给使用 SetUnhandledExceptionFilter()安装的顶层异常处理例程。如果顶层异常处理例程被安装的话，API：UnhandledExceptionFilter()将总是指向这个处理例程[1]。

接下来对 UnhandledExceptionFilter()的返回值进行检测的代码，将更深层地涉及到调试器、操作系统和 Delphi 内核之间的约定。按照这样的约定，如果当前应用在调试器环境下运行，则该函数总是返回 EXCEPTION_CONTINUE_SEARCH，一直到最后一层的异常处理程序都不响应这个异常时，操作系统将这个值发给调试器。这时调试器就会提供一个消息框[2]。

8.6.3　不使用 SysUtils.pas 单元的应用程序

如果一个应用程序不使用 *SysUtils.pas* 单元，那么当一个异常被引发之后，又会出现怎样的状况呢？操作系统仍然将控制权转给_HandleAnyException()，此后很快就会执行到如下的代码：

```
procedure       _HandleAnyException;
// ...
        MOV     EDX,ExceptObjProc
        TEST    EDX,EDX
        JE      @@exit
@@exit:
        MOV     EAX,1
// ...
```

这段代码检测系统全局变量 ExceptObjProc，如果它是 nil，则直接退出。因此，ExceptObjProc 是否赋值，是异常能否得到进一步处理的关键条件。而缺省情况下它总是 nil，要等到 SysUtils.pas 被初始化时，才在调用 InitExceptions()例程时被赋值。

因此不使用 *SysUtils.pas* 单元时，_HandleAnyException()将不做任何实质性的处理，而执行流程最终会到达_ExceptionHandler()中，并执行如下代码：

```
procedure       _ExceptionHandler;
// ...
```

① 这个处理流程是遵循操作系统对 SEH 响应机制的约定的。有关这个约定，请参考本章第 3 节。

② 调试器可能不止一次地弹出这个消息框，这取决于调试器本身的编写方法，及其对 JITEnable 和 DebugHook 两个值的响应方法。

```
        MOV     EDX, ExceptObjProc
        TEST    EDX, EDX
        JE      MapToRunError
// ...
```

MapToRunError()将一个操作系统的异常代码，映射到 Delphi 的内部错误代码上，并最终调用例程_RunErrorAt()来终止应用程序。

由于在异常处理例程_HandleAnyException()执行完之前，就调用到_RunErrorAt()终止应用程序了，所以，在不使用 *SysUtils.pas* 的应用程序中，不会真正地执行到“except notify code”，也不会执到_DoneExcept()例程。

由此可见：必须在应用程序中引用 *SysUtils.pas*，才能真正的使用 try-except-end 结构的异常处理[1]。

8.6.4 _RaiseExcept()、_RaiseAgain()与_DoneExcept()例程

以 Delphi 的代码为例，_RaiseExcept()、_RaiseAgain()两个内部例程的调用情况为：

```
procedure TestRaise;
begin
  raise Exception.Create('引发指定异常');      // call _RaiseExcept()

  try
    raise Exception.Create('引发指定异常');    // call _RaiseExcept()
  except
    raise Exception.Create('引发指定异常');    // call _RaiseExcept()

    // 重新引发正在处理的异常
    raise;                                    // call _RaiseAgain()
  end;
end;
```

_RaiseExcept()总是引发一个新的异常，这个异常对象是由在关键字 raise 后的对象实例创建代码来指定的，因此它仅需要在例程入口处检测 EAX 寄存器是否指向一个有效的对象就可以了：

① 你当然可以使用自己的单元来替代 SysUtils.pas。只要设定 ExceptObjProc 变量为一个有效的函数指针即可。当然，你可能还应该同时设定的值包括 ErrorProc、ExceptProc 和 ExceptionClass 等变量。在做这些之前，建议你参考一个很好的例子：KOL 的 *err.pas*。

```
procedure       _RaiseExcept;
asm
{ ->    EAX     Pointer to exception object     }
{       [ESP]   Error address                   }
        OR      EAX, EAX
        JNE     @@GoAhead           // 检测EAX是否为nil
        MOV     EAX, 216            // ErrCode = reAccessViolation
        CALL    _RunError           // _RunError(216)
@@GoAhead:
// ...
end
```

由于_RaiseExcept()是以 Register 约定调用的，仅一个参数(EAX 中传入异常对象)的过程，因此调用该过程时，栈顶[ESP]正好指向调用者的下一行代码，亦即是出错地址(Error address)的下一行代码。这给 Delphi 的编译器提供了一个便利，来实现 raise 关键字的另一个语法：raise ... at ...。

尽管是官方文档公开的语法规则，但 raise...at...语法并不常用，在整个的 Delphi 源码目录中，仅有 6 处使用到该语法[①]。这包括：

- 实现 ErrorHandler()用以将错误映射到异常；
- 实现 About()以触发一个“安静的”EAbout 异常；
- 在 Win32 和 Linux 环境中，为处理调用 COM 服务器的 safecall 方法中引发的 Delphi 异常而实现 SafeCallError()；
- 在 Vcl 和 Clx 中，为 TMenuItem 实现定制的错误例程 Error()。

编译器为 raise...at...语法进行特殊的编码：在调用_RaiseExcept()例程之前，首先调用 at...之后的例程，并将返回到 EAX 寄存器中的地址值入栈，然后使用 jmp 而非 call 汇编指令调用_RaiseExcept()例程。这样就模拟了直接 call _RaiseExcept()例程的堆栈现场。*SysUtils.pas* 中 About()例程的汇编代码揭示了编译器的这一实现细节：

```
ReturnAddr:
```

① 官方工具 *grep.exe* 可以协助你对批量源码进行查询和统计，该工具与 *delphi32.exe* 位于同一目录。不过使用这个工具可能需要你了解一些关于规则表达式(RegExp)的知识，这在 Delphi 帮助文件的“regular expressions”条目中有一个简略索引。在 DOS 窗口中执行如下命令行，可以得到关于 raise ... at ...的这项统计：

```
C:\>grep -r+ -i+ -d+ -c+ "raise  *[^; ]*  *at " e:\delphi7\source\*.pas
```

下面的命令行可以得到查询的详情：

```
C:\>grep -r+ -i+ -d+ "raise  *[^; ]*  *at " e:\delphi7\source\*.pas
```

```
// 在栈上取Abort()的调用者例程的返回地址
0040B2AC 8B4504                  mov eax,[ebp+$04]
0040B2AF C3                      ret

Abort:
// 调用ReturnAddr()并将返回值入栈，模拟call _RaiseExcept现场
0040B2B0 E8E7FFFFFF              call ReturnAddr
0040B2B5 50                      push eax

// 调用CreateRes()方法，返回异常对象到EAX
// 相当于EAX := EAbort.CreateRes(@SOperationAborted);
0040B2B6 8B0D5C3F4500            mov ecx,[$00453F5c]
0040B2BC B201                    mov dl,$01
0040B2BE A1986D4000              mov eax,[$00406d98]
0040B2C3 E8C4000000              call Exception.CreateRes
// 用jmp替代call，实现raise ... at ...
0040B2C8 E9BB85FFFF              jmp @RaiseExcept
0040B2CD C3                      ret
```

与_RaiseExcept()类似的，_RaiseAgain()也总是在栈上查找当前正在处理的异常并重新触发它。但这个过程要复杂很多。

异常响应句柄的执行流程在进入 except..end 块时需要恢复现场，这是通过这样的代码来实现的：

```
procedure       _HandleAnyException;
asm
// ...
        MOV     EBP,[EDI].TExcFrame.hEBP
        MOV     EBX,[EDI].TExcFrame.desc
        MOV     [EDI].TExcFrame.desc,offset @@exceptFinally

        ADD     EBX,TExcDesc.instructions
        CALL    NotifyAnyExcept
        JMP     EBX
// ...
```

这段代码表明，进入 except..end 块时：

- EBX 指向这个块的入口指针(亦即等于 EIP)；
- EBP 存放 try..except 块进入前的 EBP 值。

这也就同时表明，在进入 except..end 块时 ESP 寄存器并不改变，仍然是_HandleAnyExcpetion()执行期的值。所以在这里的用户代码只能依赖于 EBP 来访问栈，

而不能依赖于 ESP 寄存器。

ESP 寄存器指向的栈基址存放着这样一个结构：

```
procedure       _RaiseAgain;
asm
        {  ->     [ESP          ] return address to user program   }
        {         [ESP+ 4       ] raise list entry (4 dwords)      }
        {         [ESP+ 4+ 4*4] saved topmost frame                }
        {         [ESP+ 4+ 5*4] saved registers (4 dwords)         }
        {         [ESP+ 4+ 9*4] return address to OS               }
        {  ->     [ESP+ 4+10*4] excPtr: PExceptionRecord           }
        {         [ESP+ 4+10*4] errPtr: PExcFrame                  }
// ...
```

这个结构是_HandleAnyExcpetion()等异常响应例程按照 Delphi 内核实现的约定在栈上构造的。

_DoneExcept()例程也同样的依赖于这个结构：

```
procedure       _DoneExcept;
asm
        {  ->  [ESP+ 4+10*4] excPtr: PExceptionRecord            }
        {      [ESP+ 8+10*4] errPtr: PExcFrame                   }
// ...
```

有了这个结构，_RaiseAgain()要实现在栈上查找到当前被触发的异常，并重新触发它，就不难做到了。当然_DoneExcept()中的实现代码也变得很容易理解。

8.6.5　异常对象

现在应该是理清思路的时候了。

回顾前面的知识可以发现，finally 型异常是在 *System.pas* 中完整实现的，但 except 型异常仅实现了：

- 异常触发、异常重新触发、异常响应句柄、异常结束的流程；
- 在异常响应时，通过异常展开把异常传到内层的 finally 型处理句柄；
- 在栈上存放当前处理的异常列表指针、异常帧和异常记录；
- 将内部异常和 C++ Builder 模块中的异常映射成 Delphi 异常对象；
- 忽略其他未知异常(交给调试器或操作系统处理)。

应当注意到这样一个事实：现在 Delphi 已经在内核中确保要处理的异常，

- 是由其他异常映射成为异常对象的；或者，
- 是由用户通过 raise Exception.Create()这样的代码来创建的异常对象。

亦即是说，Delphi 内核只需在异常处理的过程中负责它的传输。在绝大多数情况下，“把一个异常封装成对象实例”的操作是由用户代码来完成的，而不是 Delphi 内核。而要完成“异常封装成对象实例”，就需要用户代码引用 *SysUtils.pas* 单元。

理由很简单：异常基类 Exception 被定义在该单元中。

按照内核的约定，*SysUtils.pas* 单元将初始化 *System.pas* 中的如下变量(或例程入口指针变量)[1]：

```
// code from SysUtils.pas

procedure InitExceptions;
begin
  // ...
  // 若在System.pas中异常未被处理且ExceptProc值不是nil，则将异常对象的引用和地址
  // 传入该过程
  ExceptProc := @ExceptHandler;

  // (在IDE中,)如果当前异常的基类是ExceptionClass，则调试器会启动
  ExceptionClass := Exception;

  // 在except-on..do-end型的处理中，会调用ExceptClsProc来从异常记录中检测异常类
  ExceptClsProc := @GetExceptionClass;

  // 将操作系统的未知错误映射成异常类
  ExceptObjProc := @GetExceptionObject;

  // 响应断言的例程(缺省时显示一个警告框)
  AssertErrorProc := @AssertErrorHandler;

  // 响应“调用抽象方法”异常的例程
  AbstractErrorProc := @AbstractErrorHandler;
end;
```

① 由于在 Linux 中没有 ComObj，所以在 *SysUtils.pas* 单元中，Delphi 也为支持 Linux 环境初始化 SafeCallErrorProc，它用于处理调用 COM 服务器的 safecall 方法中引发的 Delphi 异常。而在 Win32 环境中，这个变量是在 *ComObj.pas* 单元中初始化的。本书不讨论 COM 和 Linux 环境的细节。

```
initialization
  if ModuleIsCpp then HexDisplayPrefix := '0x';
  InitExceptions;

// ...
```

一旦引用 *SysUtils.pas*，except 型异常将得到处理(except..end 中的代码得到执行)。但确切地说，通常情况下的异常处理代码并不是“面向对象的异常处理”，至多可以说是在异常处理过程中“使用了对象封装机制传送异常”。例如这样的代码：

```
try
  // ...
except
  // ...
end;
```

在这种情况下内核将调用_HandleAnyException()来响应异常，这时是不识别异常类型的。而下面这样的代码[①]：

```
try
  // ...
except
  on E:Exception do // ...

  // ...
end;
```

将使用_HandleOnException()来响应，这时将识别异常的类型并分别处理。因此严格地讲，只有 except-on..do-end 型异常处理才完整地体现了“Delphi 面向对象的异常处理机制”。

异常描述记录 TExcDesc 的第二种格式[②]就是为响应 except-on..do-end 型异常而设计的：

```
TExcDescEntry =
record
  vTable:  Pointer;          // 匹配的异常类的vmt指针
  handler: Pointer;          // 如果匹配，应进入的异常处理分支的代码入口指针
end;
```

① 《Delphi 技术手册》中，这样的代码块被称为“包含异常处理块的 try-except 块”。

② TExcDesc 是变体记录，另一种格式是为 finally 型异常响应而准备的，参见本章第 4 节中的“最小化内核的启示”。

```
PExcDesc = ^TExcDesc;
TExcDesc =
packed record
  jmp: JmpInstruction;
  case Integer of
    // ...
    1{...}: (cnt: Integer; excTab: array [0..0{cnt-1}] of TExcDescEntry);
end;
```

这个结构的具体数值，是编译器根据 **except..end** 中的语法分析来得出，然后在编译期将数值直接写入代码节(这意味着它不可更改)。以这样的实例来分析：

```
procedure TryOnExcept;
begin
  try
    raise EHeapException.Create('Test!');
  except
    on E: EOutOfMemory do
      ShowMessage(E.Message);

    on E: EInOutError do
      ShowMessage(E.Message);

    on E: EHeapException do
      ShowMessage(E.Message);
  end;
end;
```

则在 IDE 的 CPU 窗口中可以看到这样的数据块：

```
// ...
00460127 EB54        jmp +$54          // Jump To Lable: OutOfExceptBlock

// ------ 以下定义了一个TExcDesc类型的记录 -------
00460129 E99A35FAFF   jmp @HandleOnException  // TExcDesc.JmpInstruction
// 下面是一个形式化的描述(字节序)①
// 数组元素数：TexcDesc.cnt
0046012E  00000003                  // TExcDesc.cnt
```

① 由于这个结构位于代码节，调试器的 CPU 窗口将会试图反汇编它，因此不可能在 CPU 窗口中看到类似这样的结果。在 CPU 窗口的数据区中，以双字(Display As DWords)方式查看这个地址，就可以看到这个结构的真实情况了。

```
// 数组: TExcDesc.excTab
00460132  00406F0C 0046014A      // excTab[0].vTable, excTab[0].handler
0046013A  00406F68 0046015A      // excTab[1].vTable, excTab[1].handler
00460142  00406EB0 0046016A      // excTab[2].vTable, excTab[2].handler
// -----------------------------------------------

// excTab[0].handler 指向这里
Unit1.pas.33: on E: EOutOfMemory do
0046014A 8945F8          mov [ebp-$08],eax
Unit1.pas.34: ShowMessage(E.Message);
0046014D 8B45F8          mov eax,[ebp-$08]
00460150 8B4004          mov eax,[eax+$04]
00460153 E848C1FCFF      call ShowMessage
00460158 EB1E            jmp +$1e

// excTab[1].handler 指向这里
Unit1.pas.36: on E: EInOutError do
0046015A 8945F4              mov [ebp-$0c],eax
Unit1.pas.37: ShowMessage(E.Message);
0046015D 8B45F4              mov eax,[ebp-$0c]
00460160 8B4004              mov eax,[eax+$04]
00460163 E838C1FCFF          call ShowMessage
00460168 EB0E                jmp +$0e

// excTab[2].handler 指向这里
Unit1.pas.39: on E: EHeapException do
0046016A 8945F0              mov [ebp-$10],eax
Unit1.pas.40: ShowMessage(E.Message);
0046016D 8B45F0              mov eax,[ebp-$10]
00460170 8B4004              mov eax,[eax+$04]
00460173 E828C1FCFF          call ShowMessage
00460178 E8B737FAFF          call @DoneExcept
Unit1.pas.42: end;

// OutOfExceptBlock :
0046017D 5F              pop edi
0046017E 5E              pop esi
0046017F 5B              pop ebx
00460180 8BE5            mov esp,ebp
00460182 5D              pop ebp
00460183 C3              ret
```

在_HandleOnException()响应句柄中，将通过对照异常实例的 vmt 来查确定是否进入相应的分支处理异常：

```
procedure       _HandleOnException;
asm
// ...
        CMP       [EAX].TExceptionRecord.ExceptionCode,cDelphiException
        JE        @@DelphiException
        CLD
        CALL      _FpuInit
        MOV       EDX,ExceptClsProc
        TEST      EDX,EDX
        JE        @@exit          // ExceptClsProc = nil ?
        CALL      EDX  // 调用ExceptClsProc, 缺省将调用SysUtils.ExceptClsProc()
        TEST      EAX,EAX         // 返回值为未知异常映射到的类引用
        JNE       @@common        // 检测返回值是否为nil. 不是则进入on..do块的类型检测
        JMP       @@exit

@@DelphiException:
        MOV       EAX,[EAX].TExceptionRecord.ExceptObject
        MOV       EAX,[EAX]                { load vtable of exception object    }

// ->   EAX中或者是未知异常映射到的类引用指针①, 或者是当前异常对象所属类的指针
@@common:
        // ...
        MOV       ECX,[EDX].TExcFrame.desc
        MOV       EBX,[ECX].TExcDesc.cnt
        LEA       ESI,[ECX].TExcDesc.excTab   // 循环列举excTab
        MOV       EBP,EAX                     // 暂存EAX

@@innerLoop:
        // ...
        MOV       EDI,EBP                     // 每次初始化EDI = EAX
        JMP       @@haveVMT

@@vtLoop:                                     // 内层循环
        MOV       EDI,[EDI]
@@haveVMT:
        MOV       EAX,[EAX]
        CMP       EAX,EDI
        JE        @@doHandler                 // 匹配成功: 类相同

        // ...
```

① 如果是未知异常映射，并不会创建一个实例，而仅是通过类引用指针来识别异常类型。

```
        // 1. 检测vmtInstanceSize
        // 2. 检测vmtClassName
        // 不匹配则跳转到 @@parent
        JE          @@doHandler                    // 匹配成功：实例长度一致且类名相同

@@parent:
        MOV         EDI,[EDI].vmtParent            // 试图访问当前异常所属类的父类...
        // ...

// ->   ESI暂存匹配的TExcDesc.excTab[n]地址
@@doHandler:
        // ...
        JMP         @@GoUnwind

@@GoUnwind:
        // ..
        PUSH        ESI                 { Save handler entry     }

        PUSH        0
        PUSH        EAX
        PUSH        offset @@returnAddress
        PUSH        EDX
        CALL        RtlUnwindProc
@@returnAddress:
        POP         EBX                 { Restore handler entry  }
        // ...

        // 跳转到on..do后的响应代码执行
        JMP         [EBX].TExcDescEntry.handler
// ...
```

在上面的这个流程中，一旦找到对异常实例进行检测类型识别时，并不访问实例本身，而是通过比较类引用的地址、InstanceSize 和 ClassName 来识别。由于 TExcDesc 是在编译期构造好的，因此在运行期的识别仅仅是非常快速的查表过程而已。

在 on..do 块中存在 else 分支，是一种很特殊的情况。这种情况下，TExcDesc 的 excTab 数组的最后一个元素的 vTable 值，将会是 nil。例如：

```
// 数组元素数：TexcDesc.cnt
0046012E    00000004                    // TExcDesc.cnt
// 数组：TExcDesc.excTab
00460132    00406F0C 0046014A           // excTab[0].vTable, excTab[0].handler
0046013A    00406F68 0046015A           // excTab[1].vTable, excTab[1].handler
00460142    00406EB0 0046016A           // excTab[2].vTable, excTab[2].handler
```

```
00460142    00000000 0046017A       // excTab[3].vTable, excTab[3].handler
```

这样，在_HandleOnException()中，就可以通过这样的一段代码来检测和响应 else 分支：

```
@@innerLoop:
        MOV     EAX,[ESI].TExcDescEntry.vTable
        TEST    EAX,EAX             { catch all clause?                 }
        JE      @@doHandler         { yes: go execute handler           }
```

8.6.6 异常类型检测的依赖性问题

在 on..do 类型的 except 处理程序中，Delphi 对异常类型的检测并不仅仅依赖于类引用的地址，同时也可以使用类实例长度和类名来识别。这样做最直接的好处是：在进程的 EXE 模块可以处理 DLL 模块中逸出的未处理异常。

但是这也可能导致更为复杂的问题。因为从 Delphi 面向对象的机制来看，仅仅识别“类实例长度和类名”，是不可能确定两个类是相同的。例如在 TryRaiseExc_DLL 模块中存在异常类 TTryException：

```
library TryRaiseExc_DLL;

uses
  SysUtils;

type
  TTryException = class(Exception)
  private
    FMsgID : Integer;
  public
    constructor Create (const Msg: string);
    property MsgID: Integer read FMsgID;
  end;


// code from IniFiles.pas, TStringHash.HashOf()
function Encode(const S: string) : Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to Length(S) do
    Result := ((Result shl 2) or (Result shr (SizeOf(Result) * 8 - 2))) xor
      Ord(S[I]);
```

```
end;

constructor TTryException.Create (const Msg: string);
begin
  inherited;
  FMsgID := Encode(Msg);
end;

procedure DoSomething;
begin
  raise TTryException.Create('Exception In DLL.');
end;

exports
  DoSomething;

begin
end.
```

而调用该 DLL 的 HOST 模块 TryRaiseExc_EXE 中也有类 TTryException:

```
program TryRaiseExc_EXE;
{$APPTYPE CONSOLE}

uses
  SysUtils;

type
  TTryException = class(Exception)
  private
    FMsgStr : String;
  public
    constructor Create (const Msg: string);
    property MsgStr: String read FMsgStr;
  end;

constructor TTryException.Create (const Msg: string);
begin
  inherited;
  FMsgStr := Msg;
end;

procedure DoSomething; external 'TryRaiseExc_DLL.dll';

procedure TestIt;
```

```
begin
  try
    DoSomething;
  except
    on E:TTryException do    // 1. 这里会混淆EXE与DLL中的TTryException类
      Writeln(E.MsgStr);     // 2. 这里输出消息时出错(存取非法地址)
  end;                       // 3. 异常处理结束时，将因错误的释放E.FMsgStr而导致出错
end;

begin
  TestIt;
end.
```

由于两个类的名字和实例长度都相等，所以 Delphi 内核会认为它们是完全相同的两个异常类。因此，在*.EXE* 的 TestIt()例程中就可以顺利进入 except..end 中的 on 分支。接下来的代码将试图把 E.MsgID 作为字符串(E.MsgStr)来读取，这将导致错误。即使 E.MsgID 很巧合地指向了一个字符串，那么一旦在异常对象实例 E 析构过程中操作字符串 E.MsgStr 的引用计数，也必然出现错误。

第 9 章

多线程的实现及调度

在 Win32 的系统构架里，进程是不执行代码的。进程是存放代码和数据的一个 4G 空间。开发人员甚至不必弄清楚为什么是 4G 这么巨大，就像旅客完全没必要弄清楚客栈是为什么、以及如何修建出那么多个房间来的一样。

为了启动代码，操作系统至少须要创建一个主线程。这个线程代码的执行入口就是 EXE 模块的入口点。如果开发人员希望在进程中“并发完成”多个操作，那么可以自由地创建任意个线程，直到系统资源耗尽。

如果进程中有一个或多个 DLL，那么 DLL 载入时并不会知道系统当前是多线程还是单线程。但 DLL 被载入后，在每创建一个新线程时都会收到一个消息，即使新线程根本就不调用 DLL 中的代码。

绝大多数 Win32 API，以及 Win32 应用程序所有缺省载入的 DLL 都默认进程是在单线程方式下运行的，Delphi 的内核函数和 VCL 也这样认为。因此如果在一个进程创建了多个线程，则编写代码时必须兼顾到这一点。对于多个线程都要访问的数据或系统资源，必须确保同时只有一个线程在占用和访问。有一些特殊的情况下，也可以选用“多读单写”方案。

本章将从操作系统的多线程机制入手，详细讲述 Delphi 的内核为多线程应用的编写实现了哪些机制，如何用线程类来封装和管理线程，以及用户代码如何在多线程中实现同步。本章的两个重点是线程局部存储和线程类。

Delphi 内核中用于实现多线程的代码并不多。通常情况下，Delphi 的应用是运行于单线程模式的，而且，对于大多数数据类型(包括类、接口等)，都是非线程安全的。

可以直接使用 API 或 Delphi 的例程来安全地实现多线程，但如果这样在应用中加入多线程

支持，就需要自行管理线程同步对象。大多数情况下，可以使用如下的线程同步对象类：

- TCriticalSection。
- TEvent。
- TSimpleEvent。
- TSimpleRWSync。
- TThreadLocalCounter。
- TMultiReadExclusiveWriteSynchronizer(TMREWSync)。

或者直接使用 API 来管理同步对象。此外，与多线程管理相关的概念还包括“会话”、“线程池”和“线程列表”。

与上述这些内容相关的内核实现包括下列内容：

- 线程变量(线程局部存储)。
- 线程的创建与释放。
- 线程同步对象。
- 多线程管理。
- 多线程环境下的内存管理器。

9.1　操作系统的进程与线程[①]

无论是在资源管理器中双击可执行程序，还是通过快捷方式或命令行启动它，系统都是调用 CreateProcess()来启动它。在开始执行这个可执行程序中的第一行代码之前(EXE 模块的入口点)，操作系统会做大量的前期工作。

这些工作中相当重要的部分就是创建一个进程空间，并把待启动的 *EXE* 模块载入到这个进程空间。载入操作最初仅是做一个内存地址空间的分配，并将 PE 文件头中的各个节数据读入到内存空间中或进行简单的地址映射。载入操作的结果是使得从表面看来：进程的 4G 空间中的确有一个该 *EXE* 模块的完整映像。至于文件数据是否读入物理内存或物理存储中，抑或根本就是在内存地址与文件读取动作之间建立了一个关联[②]，则是开发人员不须要关心的。

① 本书并不试图揭示操作系统的内幕。但为了便于行文，部分相关章节中会概要描述操作系统的一些行为。有关这些行为的内部实现的进一步探索，建议阅读本书附录中的参考书籍。

② 操作系统并不一定将 PE 文件的全部节读入到物理内存或者内存页文件。一个地址映射，可能表明仅仅在直接存取该节数据时，操作系统才会打开文件，将数据读到内存。

除.*EXE*模块本身之外，第一个被映射到进程空间的模块是 *NTDLL.DLL*。随后操作系统开始为进程创建一个主线程，并在进程空间中创建用户态的堆栈。在完成一些与线程有关的内核数据结构的初始化操作之后，主线程被挂起。

接下来，系统在核心模式下为进程和线程的创建发出消息、调整计数和设置标志，同时附加、异常等与代码执行相关的结构也被创建。

在此之后，主线程开始启动。

9.1.1 线程的启动过程

线程被创建的初期处于挂起状态。对于操作系统来说，线程就是一些诸如描述表、环境块等数据结构的指针，这些指针存放于 ETHREAD(Executive Thread，执行体线程)块结构中。除了 ETHREAD 和它指向的结构之外，线程本身没有数据体，这与进程不同。线程是执行代码的一个控制结构，而非代码 / 数据的承载体(容器)。

ETHREAD 块所指向的所有结构中，除了 KTHREAD 结构中的 TEB(Thread Environment Block，线程环境块)位于进程地址空间内，其他的结构都位于系统地址空间内[1]。

ETHREAD 块中存放与代码执行直接相关的两个地址是 KiThreadStartup()例程的入口和线程入口代码地址。其中 KiThreadStartup()来自于操作系统的内核 NTOSKRNL。而线程入口代码地址存在两种情况：

- 如果是主线程，这个地址是 EXE 模块的 PE 结构中的入口点；
- 如果是用户线程，这个地址是用户代码在调用 CreateThread()时指定的 lpStartAddress 参数。

在主线程被唤醒前，第一段为当前进程执行的代码将是 KiThreadStartup()例程，这个例程将在操作系统进程中被执行。

KiThreadStartup()例程会把一个进程所需的其他全部内核代码映射到进程空间[2]。此外还将分析该.*EXE*模板的导入表，来确定应当将哪些相关的静态 DLL 同时载入内存和建立映射。对导入表的分析将扩散到所有静态载入的 DLL 模块，确保位于这个庞大的模块树中的所有模块都被加载。

(1) 这两种地址空间的区别涉及到操作系统的内存机制，请参阅相应的书籍。

(2) 操作系统的内核代码是进程间共享使用的，例如多个进程间共享使用同一个 *ntdll.dll*。这充分利用了操作系统定义的 DLL 文件载入规则，尽管并不是每个进程都拥有一份 *ntdll.dll* 的代码拷贝，但在进程看来，进程地址空间的某个确定位置总是存在这个 *ntdll.dll* 的一个映像。这是在操作系统的内核中通过地址映射来实现的。

进程所需的内核代码的绝大部分被称为“Win32 子系统的 DLL”，这包括 *USER32.DLL*、*KERNEL32.DLL* 和 *GDI32.DLL* 等等这些模块。一个常见的清单如下：

```
        Base     ImageEnd   Path
------------------------------------------------------------------------
ModLoad: 77e40000 77f4e000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 77d10000 77d9c000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77c40000 77c80000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 77da0000 77e3b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 78000000 780b6000   C:\WINDOWS\system32\RPCRT4.dll
```

如果内核要加载 *USER32.DLL*，则系统会在处理 DLL_PROCESS_ATTACH 的过程中，访问如下注册表项并加载其中指定的 DLL 模块：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\
AppInit_DLLs
```

加载 APPTYPE 类型为控制台应用程序的内核时，不载入 *USER32.DLL* 的情况的确是可能发生的。不过使用 Delphi 的程序员几乎没有机会看到。因为 Delphi 的 *System.pas* 所初始化的 RTL 环境都将引用 *USER32.DLL* 模块——尽管实际代码中可能并不调用 *USER32.DLL* 中的 API。而在使用最小化内核编译的控制台程序中就可以看到这一现象了，例如运行在随书光盘中的 *MiniConsole.exe* 并使用 listdlls 工具查看：

```
C:\>listdlls MiniConsole
ListDLLs V2.2: - DLL lister for Win9x/NT
Copyright (C) 1997-2000 Mark Russinovich
http://www.sysinternals.com

------------------------------------------------------------------------
MiniConsole.exe pid: 1712
Command line: MiniConsole

  Base            Size        Version          Path
  0x00400000      0x7000                       E:\Console\MiniConsole.exe
  0x77f30000      0xca000     5.02.3790.0000   C:\WINDOWS\system32\ntdll.dll
  0x77e10000      0x11d000    5.02.3790.0000   C:\WINDOWS\system32\kernel32.dll
```

一旦内核加载成功，主线程就真正地开始启动[①]，在进程中载入以下两份 DLL 模块清单，并逐一调用这些 DLL 的入口点。

① 如果进程在调试环境中运行，那么在启动线程之前，操作系统会向调试器会发送一个 CREATE_THREAD_DEBUG_INFO 的线程启动调试事件。

第一个 DLL 模块列表可能总是会被加入到进程中，这些模块用于初始化输入法、本地化设置，以及载入需要用到的运行时间库(RTL)，通常包括：

```
          Base      ImageEnd  Path
-------------------------------------------------------------------
ModLoad: 76300000 76310000  C:\WINDOWS\System32\IMM32.DLL
ModLoad: 62c30000 62c28000  C:\WINDOWS\System32\LPK.DLL
ModLoad: 72f10000 72f6a000  C:\WINDOWS\System32\USP10.dll
ModLoad: 77bd0000 77bd7000  C:\WINDOWS\system32\version.dll
ModLoad: 003b0000 003db000  C:\WINDOWS\System32\msctfime.ime
ModLoad: 77be0000 77c33000  C:\WINDOWS\system32\msvcrt.dll
ModLoad: 7cab0000 7cb61000  C:\WINDOWS\System32\ole32.dll
ModLoad: 74650000 74676000  C:\WINDOWS\System32\MsImtf.dll
ModLoad: 74680000 746c4000  C:\WINDOWS\System32\MSCTF.dll
```

第二份加载的模块列表，才是来自*.EXE* 模块 PE 结构(导入表中)所记录的引用 DLL 列表。

最后系统将转入对主线程入口地址上的代码的调用。对于进程创建来说，这意味着将调用 EXE 模块的 PE 结构中的入口点。

只有在必要的时候，操作系统会将其他进程钩挂的全局 HOOK DLL 模块强制注入到该进程中。没有任何文档提及全局 HOOK 一定会在进程入口代码执行前被注入，全局 HOOK 的注入时间与它的类型和该 HOOK 的初始代码直接相关。

9.1.2 主线程与用户线程启动过程的比较

系统创建的主线程与普通线程并没有本质的不同，但主线程创建时的入口参数 lpParameter 指向由进程创建时初始的 PEB(Process Environment Block)，dwStackSize 则取自 PE 文件头中对初始栈空间的设定。而在普通线程中，这两个值是调用 CreateThread()时传入的。

KiThreadStartup()是位于核心态的系统范围内的启动例程。它不是 *NTDLL.DLL* 中的代码。它总是在核心模式下运行，这无论是对用户线程还是主线程来说，都是一样的。但是在 KiThreadStartup()调用过程中，将遍历进程模块树上的每一个 DLL，并以 DLL_PROCESS_ATTACH 为参数来调用 DLL 入口点。如果是在用户线程中，则还将用 DLL_THREAD_ATTACH 调用 DLL 入口点。在调用这些入口点时，系统将切换到用户模式下执行。

操作系统没有以 DLL_THREAD_ATTACH 来调用静态 DLL 的入口点，是很容易理解的：进程加载静态 DLL 的操作，位于主线程创建之后。

DLL 模块静态载入时只接收 DLL_PROCESS_ATTACH 为参数的 DllMain 调用。而作为动态模块加载[1]时将会收到 DLL_PROCESS_ATTACH 和 DLL_THREAD_ATTACH 两个参数的两次 DllMain 调用，这可以作为在 DLL 中识别自己被载入方式的标志。由于每次新建线程时 DLL 都会收到以 DLL_THREAD_ATTACH 为参数的 DllMain 调用，因此在 DLL 中能够实时地识别当前系统是处于多线程还是单一线程模式[2]。

DLL 作为静态模块载入时还具有一项极重要的特性：内核已经映射到进程，且进程已经初始化，但主线程还没有调用 EXE 模块的入口点。因此在这时，DLL 模块仍有机会调整 EXE 初始前的一些参数。举例来说，在 DLL 中，可以加入动态的 PATCH 代码来修改 EXE 模块中的全局常量、变量或者函数入口，甚至可以重新(在内存中)修正 EXE 模块的导入导出表。这些在 DLL 静态模块中执行的代码，都是在进程的主线程(用户模式)中执行的，因此具有与 EXE 模块完全一致的执行权限和存取范围[3]。

9.1.3　线程有哪些独立元素

每个线程在通常情况下都具有两个栈[4]：

- 内核模式下的堆栈，位于系统地址空间，相关信息记录于 ETHREAD 结构中；
- 用户模式下的堆栈，位于进程的用户地址空间，相关信息记录于线程的 TEB 结构中。

同一线程的代码执行过程中，可能在内核模式和用户模式间不断切换。例如应用程序调用 API：CloseHandle()，在系统执行到 ntdll.dll!NtClose 时会执行 INT 2E 指令。一旦 2E 中断发生，系统在进入中断处理程序 *ntoskrnl!KiSystemService* 之前会将当前特权级从 ring3 变为 ring0，这时线程堆栈也会从用户模式转为内核模式下的堆栈。一个显著的变化是：ESP 和 EBP 寄存器改变了。

操作系统为每个线程保存一组独立的寄存器数据和状态信息。由于寄存器 FS:[0]中保存着 SEH 控制结构的指针，因此这也意味着每个线程都有独自的 SEH 控制结构。

另一个明显属于线程的独立元素是“线程局部存储(TLS)”。

[1] 使用 LoadLibrary()或者 LoadLibraryEx()函数。

[2] 的确有 API 来做类似的事。但这里主要是想说明多线程模式对进程中不同模块的影响。

[3] 我发布的开源项目 ShareMemRep 中就充分利用了静态模块的这一特性。请参见光盘中的相关代码。

[4] 某些只在内核模式运行的线程(如 System 进程的一些线程)没有用户模式堆栈。

9.2 操作系统的线程调度概要

首先，并不存在“进程调度”的概念。进程不是执行体，并不需要“调度”。操作系统的调度策略严格地以线程为基本单位进行，而并不考虑被调度线程是属于哪个进程。

大多数情况下，(进程优先级相同的)多个线程的每一线程分到的 CPU 时间是一致的。但这并不绝对。进程只能有一个优先级值(基本优先级)，而线程却有两个优先级值：

- 基本优先级：继承自进程基本优先级；
- 当前优先级：通常等于或高于基本优先级。[1]

如果线程中正在执行的代码不能被安全地暂停(例如硬件中断响应例程)，那么操作系统通常不会强行地将 CPU“挪作它用”。这是因为所有线程优先级都是在中断优先级中处于很低的一个位置如图 9.1 所示：

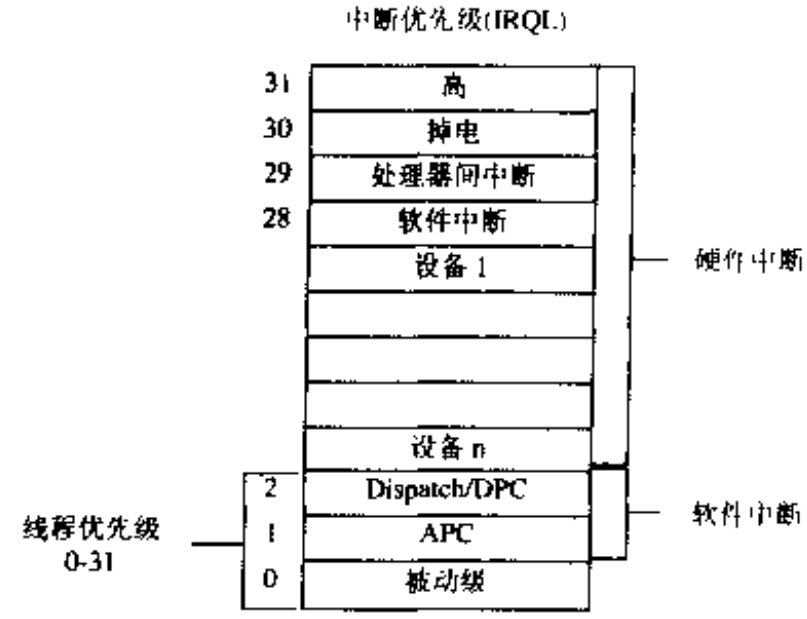


图 9.1 线程优先级在中断优先级中所处的位置

此外，在实时优先级(16~31 级)范围内运行时间过长的线程，还有可能封锁掉在内存管理器、高速缓冲管理器、本地和网络文件系统、甚至其他设备驱动程序中对关键系统函数的调用。

这些都可能使得一些线程的执行时间远比预想的要长。例如开发人员在一个线程使用代码 `sleep(1000)`来让出了 CPU 控制权，那并不意味着当前线程会在一秒钟后准时地再次得到 CPU 控制权。在“无法确知 `sleep()`时间的长度”的前提下，`sleep(1)`与 `sleep(1000)`可

① 如果优先级在动态范围内(1~15 级)，则线程的当前优先级可能(通常总是)高于基本优先级；如果处于实时范围内(16~31 级)，则该线程的优先级不会被系统自动调整，因此当前优先级总是等于基本优先级。

能的实际效果是一样的：仅表明当前线程让出对 CPU 的占用。

上面的文字试图说明 Windows 多线程调度机制的几点事实：

- 线程执行的时间并不一定如开发人员在代码中预期(或预设)地那样短或者固定；
- 线程休眠的时间并非精确的；
- 线程什么时候会再开始执行是不可预期的。

此外须要知道的是：线程休眠并不表明在线程中控制的外部设备也会随之停止工作[①]。

9.3　基本的多线程应用

可以使用 API 来开启一个多线程，并执行一个例程。直到例程结束、线程中止(TerminateThread)或者线程退出(ExitThread)之前，操作系统会维护这个线程的状态及其局部存储。通常会用到的 API 有：

```
// 创建一个新线程
function CreateThread(SecurityAttributes: Pointer; StackSize: LongWord;
  ThreadFunc: TThreadFunc; Parameter: Pointer; CreationFlags: LongWord;
  var ThreadId: LongWord): Integer; stdcall;
  external kernel name 'CreateThread';

// 从当前线程中退出
procedure ExitThread(ExitCode: Integer); stdcall;
  external kernel name 'ExitThread';
```

直接使用这些 API 来实现多线程的代码非常简单：

```
function ThreadFunc(Parameter: Pointer): Integer; stdcall;
begin
  Writeln('Hi, in MultiThread ...');

  // ...
  // 在线程函数中，随时可以调用ExitThread()来退出
```

① 这句话似乎多余，但我的确遇到过这样的情况：一位程序员将打印机因其他意外停止工作的问题，归咎于使用了多线程。于是他修改 BUG 的结果，是使那个小应用程序再也无法支持“后台打印”。更加要命的是，BUG 依旧一如既往地出现。

```
  // 或者当代码执行到函数结束关键字时，也将退出该线程
end;

var
  ThreadID : Cardinal;
  ThreadHandle : Cardinal;
begin
  ThreadHandle := CreateThread(nil, 0, @ThreadFunc, nil, 0, ThreadID);

  //...
  // 在线程函数之外，可以随时调用TerminateThread(ThreadID, ThreadHandle)
  // 来中止线程
end.
```

除了多线程对象 TThread 之外，Delphi 也提供一些内核例程用于开始和结束多线程。这主要包括：

```
type
  TThreadFunc = function(Parameter: Pointer): Integer;

// 开始一个线程
function BeginThread(SecurityAttributes: Pointer; StackSize: LongWord;
  ThreadFunc: TThreadFunc; Parameter: Pointer; CreationFlage: LongWord;
  var ThreadId: LongWord): Integer;

// 结束一个线程
procedure EndThread(ExitCode: Integer);
```

这两个例程都是调用对应的两个操作系统 API：CreateThread()和 ExitThread()。略有不同的是在例程 BeginThread()的代码中，有另外两项操作：

- 设全局变量 IsMultiThread 为 True[①]；
- 在堆中初始化一个 ThreadRec 记录，并作为参数调用 CreateThread()。

BeginThread()并不直接向 CreateThread()传入 ThreadFunc 参数。这个参数被填入 ThreadRec 记录中，然后和 ThreadWrapper()例程一同传入 API：CreateThread()。

ThreadWrapper 在用户的 ThreadFunc 之外封装一层代码。这层代码要实现的仅仅是线程一级的异常管理机制：

① 参见本章第 8 节“多线程环境下的内存管理器”的描述。

```
function ThreadWrapper(Parameter: Pointer): Integer; stdcall;
asm
        CALL    _FpuInit
        PUSH    EBP

        // 1. 将一个异常帧记录添加到FS:[0]的链表表首
        XOR     ECX,ECX
        PUSH    offset _ExceptionHandler
        MOV     EDX,FS:[ECX]
        PUSH    EDX
        MOV     FS:[ECX],ESP


        // 2. 入口参数Parameter是TThreadRec记录，将记录中的参数(Parameter)和线程
        // 例程入口(Func)压栈暂存，然后释放TThreadRec记录。
        MOV     EAX,Parameter
        MOV     ECX,[EAX].TThreadRec.Parameter
        MOV     EDX,[EAX].TThreadRec.Func
        PUSH    ECX
        PUSH    EDX
        CALL    _FreeMem

        // 3. 以Register调用约定CALL TThreadRec.Func()
        POP     EDX
        POP     EAX
        CALL    EDX

        // 4. 汇编语句"CALL EDX"在当前线程中执行，一旦线程例程执行结束，将进入下面的
        // 代码，开始异常帧恢复和堆栈恢复，随后操作系统将结束线程。
        XOR     EDX,EDX
        POP     ECX
        MOV     FS:[EDX],ECX
        POP     ECX

        POP     EBP
end;
```

Delphi 引入 ThreadRec 记录的目的仅是为了暂存线程创建的参数和例程入口地址。应当注意到 API: CreateThread()是在父线程中调用的，而 ThreadWrapper()却在被创建的新线程中执行。这两个线程使用各自独立的栈，所以 BeginThread()不能通过栈来向 ThreadWrapper()传递数据。又因为两个线程在同一进程空间，因此可以传入记录指针并在新线程中 FreeMem()。

Delphi 源代码分析

9.4 线程变量(线程局部存储)

Delphi 中所声明的 threadvar 总是被存放在线程局部存储(TLS)中[①]。在线程环境块(TEB)中的域 pvTLSArray 总是指向线程局部存储数组。

一旦有线程变量被定义，则编译器将为每次访问提供一个偏移值(基于当前模块的 TLS 的字节索引)。因此一个使用了线程局部变量的代码通常会编译生成这样的代码：

```
// 源代码
threadvar
  TlsStr : String;
  TlsInt : Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  TlsInt := 3;
  TlsStr := 'abcd';
end;

// 编译后
Unit1.pas.31: TlsInt := 3;
0044C879 E8FF92F8FF          call @GetTls
0044C87D C7801400000003000000 mov [eax+$00000014],$00000003
Unit1.pas.32: TlsStr := 'abcd';
0044C887 E8F092F8FF          call @GetTls
0044C88C 8D8010000000        lea eax,[eax+$00000010]
0044C892 BAA8C84400          mov edx,$0044c8a8
0044C897 E8C875F8FF          call @LStrAsg
Unit1.pas.33: end;
//...
```

在这个示例中，TlsInt 使用的偏移为$00000014，而 TlsStr 使用的偏移为$00000010。从例程 GetTls()中返回的 TLS 起始地址存放在 EAX 寄存器中。

编译器并不优化掉对 GetTls()的多次调用，这与 Tls 存取机制有关。这个例程在 *SysInit.pas* 中实现为：

① 这是不很确切的说法。threadvar 是 Delphi 的一个语法关键字，而线程局部存储是一种操作系统的机制。二者并不处于同一层面。它们之间是“定义”与“实现”的关系。

```
function _GetTls: Pointer;
asm
        MOV       CL,ModuleIsLib
        MOV       EAX,TlsIndex
        TEST      CL,CL
        JNE       @@isDll
        MOV       EDX,FS:tlsArray
        MOV       EAX,[EDX+EAX*4]
        RET

@@initTls:
        CALL      InitThreadTLS
        MOV       EAX,TlsIndex
        PUSH      EAX
        CALL      TlsGetValue
        TEST      EAX,EAX
        JE        @@RTM32
        RET
@@RTM32:
        MOV       EAX, tlsBuffer
        RET

@@isDll:
        PUSH      EAX
        CALL      TlsGetValue
        TEST      EAX,EAX
        JE        @@initTls
end;
```

9.4.1　线程局部存储(TLS)的实质

无论如何，线程局部存储也仅仅是一块内存。这块内存使用一个索引来访问，而 API：TlsAlloc()则用于在进程范围内为线程分配这个索引值。如果一个模块从来没有调用过这个 API，那么也表明这个模块并不使用 TLS[①]。

应该奇怪的是：这里是“模块与 TLS”之间的关系，看起来与线程无关。但应该清楚的是：编译器并不知道“线程”的存在。也就是说，必须有一种方法，来使得编译器实现一种在运行

① 从数据结构层面来讲，TLS 指的是一块物理存储(memory)；从操作系统实现上来讲，TLS 指的是一种机制。大多数情况下，TLS 这个词汇是被混用的。但在接下来的讨论中，将会使用“TLS 块”来表示内存，而 TLS 则表示整个机制的名称。

期将“线程变量地址”与线程关联起来的机制。

编译器只能在模块中为“线程变量”写下一个固定的地址，这个地址必须能在不同的线程中得到修正，只有这样才能实现 TLS。反过来从操作系统的角度来看，如果一个模块不使用“线程变量”，那么也就不需要为这个模块准备与 TLS 有关的任何东西(例如在 pvTLSArray 中分配一个指针)。

编译器为线程变量在模块中存放提供的地址是一个基于“TLS 块”的偏移值，如果能保证在不同线程进入该模块中执行代码时，给出一个基地址不同、大小相等的“TLS 块”，那么通过地址运算所访问到的变量值就与线程相关了。

这就是 TLS 机制的实质。实现这样的机制，需要操作系统、编译器与 Delphi 内核代码的交互协作，这包括：

- 编译器在模块中记录每个线程变量相对于 TLS 块首的偏移；
- 对每个线程，操作系统保证在每次进入线程时，给线程一个独立的 pvTLSArray 指针，所指向的数组用于存放 TLS 块首地址；
- 对于每个使用 TLS 的模块，操作系统与 Delphi 内核代码保证每个线程都有一个独立的 TLS 块；
- 编译器与内核代码保证每次操作线程变量时，都使用它在 TLS 块中的偏移值来存取。并且在不同线程中，使用相同的 pvTLSArray 索引值来获取 TLS 块首地址。

9.4.2 编译器的准备工作

在本小节开始的示例中，已经很明确地指出“编译器计算出线程变量相对于 TLS 块的偏移，并保证每次操作变量时，都使用该偏移值来取值”：

```
Unit1.pas.37: TlsStr := 'abcd';

// 总是调GetTls()来获取TLS块首地址到EAX中
0044C887 E8F092FBFF      call @GetTls

// $00000010是编译器运算并作为立即数编码
0044C88C 8D8010000000    lea eax,[eax+$00000010]
0044C892 8AA8C84400      mov edx,$0044c8a8
0044C897 E8CB75FBFF      call @LStrAsg
```

在这段代码的最后一行中，由于 EAX 已经存放了 TlsStr 的指针，所以调用_LstrAsg()时与访问普通变量并没有区别(例如在 EAX 值为 nil 时，则向 EAX 存入 EDX 所指向的字符串的引用或一个新串的地址)。

除此之外，为了配合内核代码来初始化 TLS 块，编译器还将做两件重要的事：

- 定义模块的全局变量 TlsIndex，用于访问 pvTLSArray；
- 定义模块的全局变量 TlsLast，用于保存模块使用 TLS 块的大小。

虽然 TlsLast 被定义成一个 Byte 类型的变量，但它实际上只是一个占位的符号而已。永远不可能直接访问这个变量来得到一个 Byte。事实上，这可能导致内存非法存取。

编译器总是使用@TlsLast 来得到一个整型的 TLS 块大小值，也总是将该值作为立即数直接编码。在汇编码一级，这与使用一个常值(立即数)并没有什么不同。但是 Delphi 需要在模块中导入 TlsLast 的地址，因此它必须定义一个变量来骗过语法解释器——普通常量是没有地址可以导出的。

```
// 示例代码:不要试图直接存取TlsLast!
var
  TlsSize : integer;

begin
  TlsSize := Integer(@TlsLast);
  // ...

// 反汇编代码
Project2.dpr.29: TlsSize := Integer(@TlsLast);
00414693 B810000000    mov eax,$00000010   // 编译器总是直接将长度值作为立即数编码
00414698 A37C664100    mov [TlsSize],eax
```

另一个非常关键的全局变量是 TlsIndex。该变量用于存放“TLS 时隙[①]”数组(TLSArray)的索引值。这是操作系统为每线程准备的一份表。在 Windows 2000 以下的操作系统中，该索引的范围为 0~63，在 Windows 2000 以上的操作系统中，则允许设置该值超过 1000。

TlsIndex 值仅对使用 TLS 的模块有意义。很显然，由于 TlsIndex 的值范围有限，所以在同一进程中，能同时载入的、使用 TLS 的模块也将是有限的——这取决于操作系统的约定，而与 Delphi 无关。

事实上，操作系统也仅仅在进程中为多线程维护 TLSArray 表。Delphi 使用一个常量来指向该域在 TEB 中的偏移：

```
const
```

① 这个古怪的名词来自于《WINDOWS 核心编程》，一些资料也翻译作“线程局部存储时间片”，对应的英文通常是“TLS Slot”。本书中通常用 TLSArray 来表明存放它的数组，而将每一个数组元素称为“TLS 块”。数组大小决定了同一进程中可以同时使用 TLS 的模块数量。

```
  tlsArray      = $2C;    { offset of tls array from FS: }
```

TLSArray 的每一数组元素是一个“TLS 块”首地址指针。同一进程中，极限情况下，“TLS 存储块”的数量等于“线程数”与“使用 TLS 机制的模块数”的乘积。对于如下的代码：

```
// 源代码:
threadvar
  iData : integer;
begin
  iData := 5;
  //...

// 编译后(用 TlsIndex 在 TlsArray 中查找TLS块基地址到EAX):
Module2.dpr.27: iData := 5
00414675 E8FB14FFFF              call @GetTls
0041467D C7800C0000000500        mov [eax+$0000000C],$00000005
```

图 9.2 说明在不同线程中存取 iData 时进行的地址操作。

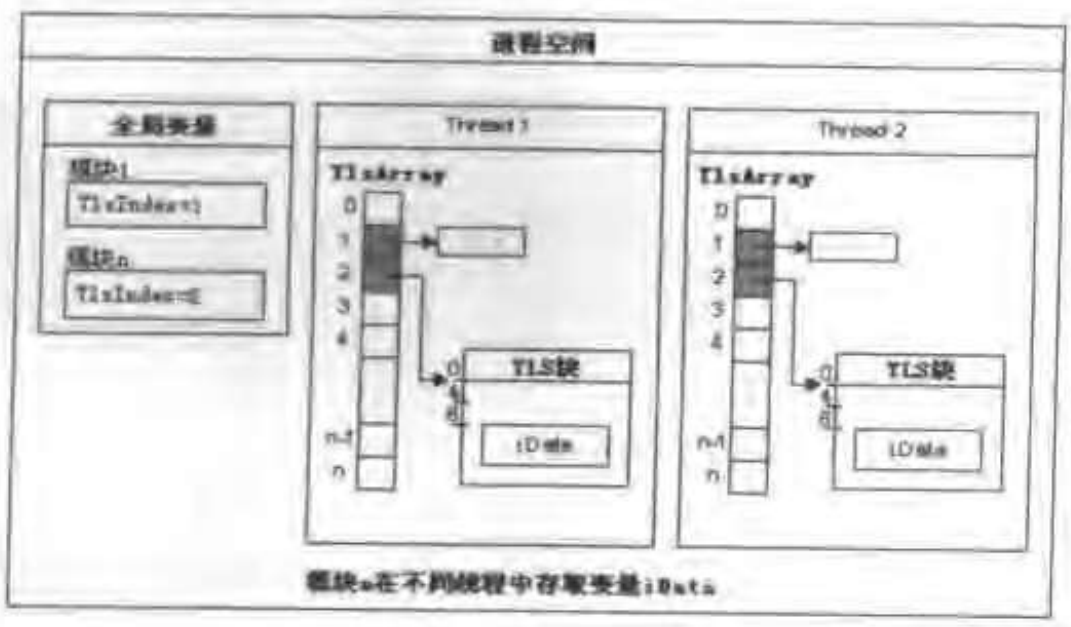


图 9.2 在不同线程中存取 iData 时进行的地址操作

9.4.3 静态 TLS：EXE 模块中的 TLS

操作系统为.*EXE* 模块创建了主线程之后，EXE 模块不会再得到任何关于线程创建的消息。这也就是说，.*EXE* 模块没有响应操作系统的线程创建动作的有效机制，因此操作系统不得不单独管理 EXE 模块所需用的 TLS 块。

由操作系统针对模块直接管理 TLS 的物理存储的方式叫静态 TLS，这是在.*EXE* 模块中使用 TLS 的惟一方式。

通常情况下，Delphi 生成的 EXE 中会有 TLS 节。这个节在 EXE 模块中并没有实际数据。例如：

```
C:\>tdump E:\Delphi7\Bin\serverinfo.exe
#   Name      VirtSize   RVA      PhysSize  Phys off  Flags
--  --------  --------  --------  --------  --------  --------
01  CODE      0005CEA0  00001000  0005D000  00000400  60000020 [CER]
02  DATA      00001F10  0005E000  00002000  0005D400  C0000040 [IRW]
03  BSS       00000C69  00060000  00000000  0005F400  C0000000 [RW]
04  .idata    000017A0  00061000  00003000  0005F400  C0000040 [IRW]
05  .tls      00000018  00065000  00000000  00062C00  C0000000 [RW]
//...
```

操作系统在载入 EXE 时，会查看这个节区入口，并依据 VirSize 值来创建 TLS 块，块首地址被记入主线程的 TEB 的 pvTLSArray 所指向的数组的第 0 个元素中。同样，操作系统在每创建一个新线程时，也再次创建这样大小的一个 TLS 块，并在新线程的 pvTLSArray 表的元素 0 中记录块首地址。

这样，Delphi 就只需要在 EXE 模块的初始化中置 TlsIndex 值为 0 即可。

```
procedure _InitExe(InitTable: Pointer);
begin
  TlsIndex := 0;
  // ...
end;
```

9.4.4　动态 TLS：DLL 模块中的 TLS

对于操作系统来说，使用动态 TLS 的开销是很大的。因此 Delphi 在 DLL 中并不使用动态 TLS 机制，而是在内核代码一级实现了动态 TLS 机制来管理 TLS 块。

使用动态 TLS 时，操作系统并不准备任何线程变量的具体数据，也不分配它的存储空间。所以，在系统以 DLL_PROCESS_ATTACH 为参数调用 DllMain() 入口点(这是该 DLL 模块被执行的第一行用户代码)之前，没有任何线程知道该 DLL 是否使用 TLS 机制。

在 DLL 模块中，系统使用 TlsProc 数组来定义 DllMain() 不同入口时所调用的内部例程：

```
const
  TlsProc: array [DLL_PROCESS_DETACH..DLL_THREAD_DETACH] of procedure =
    (ExitProcessTLS,InitProcessTLS,InitThreadTLS,ExitThreadTLS);
```

在 InitProcessTLS() 中将通过调用 TlsAlloc() 来初始化 TlsIndex 的值。(在同一进

程中，)由于每个 DLL 模块只接受到一次 DLL_PROCESS_ATTACH 和 DLL_PROCESS_DETACH 的调用，因此也就意味着每个 DLL 会占用一个 TlsIndex 索引号，且直到该 DLL 模块从进程中卸载时，才会调用 ExitProcessTLS()来释放它。

在以 DLL_THREAD_ATTACH 调用 DllMain()时，还将使用 InitThreadTLS()例程来复核模块中 TlsIndex 的有效性：

```
procedure       InitThreadTLS;
var
  p: Pointer;
  tlsSize: Integer;
begin
  tlsSize := GetTlsSize;                          // 当前模块使用TLS存储空间的长度
  if tlsSize = 0 then  Exit;                      // 使用TLS吗?

  // 如果使用TLS，但TlsIndex又未被初始化，则出错。
  if TlsIndex = -1 then  RunError(226);
//...
```

一旦得到有效的 TlsIndex，当前线程就可以在 TLSArray 中查到 TLS 块的首地址。其实现代码如下：

```
function _GetTls: Pointer;
asm
        MOV       EAX,TlsIndex
        // ...

        MOV       EDX,FS:tlsArray
        MOV       EAX,[EDX+EAX*4]
        RET
//...
```

在 *SysInit.pas* 中引用到的与动态 TLS 有关的四个 API 是：

```
// 为当前模块分配一个TlsIndex
function TlsAlloc: Integer; stdcall;
  external kernel name 'TlsAlloc';

// 释放TlsIndex
function TlsFree(TlsIndex: Integer): Boolean; stdcall;
  external kernel name 'TlsFree';

// 从当前线程的TLSArray中，取由TlsIndex指定的元素中的TLS块地址
function TlsGetValue(TlsIndex: Integer): Pointer; stdcall;
```

```
  external kernel name 'TlsGetValue';

// 如果当前线程尚未对指定模块分配TLS块，则用TlsValue指向一个新分配内存，
// 将TlsValue指向的TLS块首地址记入TLSArray，
function TlsSetValue(TlsIndex: Integer; TlsValue: Pointer): Boolean;
  stdcall; external kernel name 'TlsSetValue';
```

在例程 TlsSetValue()中，并没有约定 TlsValue 所指向的内存块来自于进程内存地址的哪个区间。因此，Delphi 为了可以在 *System.pas* 单元中不调用自己的 MemMgr 分配内存[①]，就使用 LocalAlloc()在进程堆上分配内存作为 TLS 块，并填充零值[②]：

```
const
  LMEM_ZEROINIT = $40;

function AllocTlsBuffer(Size: Integer): Pointer;
begin
  Result := LocalAlloc(LMEM_ZEROINIT, Size);
end;
```

没有任何 Tls 函数可以在 A 线程中操作 B 线程的 TLSArray 并存取数据。如果要使两个线程之间有数据交互，那么可以使用进程中的全局变量，或像 ThreadWrapper()例程一样传入一个内存指针，都是不错的解决方案。

9.5　再论 FS 段

FS 段与 FS[0]一再地在 *System.pas* 中出现，而且在 CPU 窗口中也有着与 SS、DS 等段寄存器不一样的值。这不由得让人产生疑问：FS 段到底是什么？

FS 段是在(每)一个线程创建时，由操作系统来分配内存并向 FS 寄存器置值的。准确地说：

- 当程序运行在用户模式下，FS 段总是指向当前线程的 TEB 块所在地址空间；FS[0]指向 TEB 块首；
- 当程序运行在内核模式下，FS 段指向一个内核数据块的地址空间。通常这个数据块开始于$FFDFF000，大小为$2000。

没有办法在 Delphi 的 CPU 窗口中看到 FS 段寄存器(选择器)所指向的实际虚地址。在没有

① 如果 *System.pas* 中使用了 GetMem.inc 中的内存分配函数，将导致用户的第三方内存管理器无法安全加载。
② 这样可以避免 Delphi 将一个无意义的值作为字符串引用处理。

类似 SoftIce 这样的工具的情况下，可以通过计算来得到这个地址值[①]。

下面以一个新建的 Delphi 应用 *Project1.dpr* 运行期在 CPU 窗口中的值为例。在源代码中设一个断点，当 IDE 在该断点中断时，打开 CPU 窗口，查看寄存器值如下（图 9.3）。



图 9.3 查看寄存器值

记下寄存器值：

```
FS: 0038
DS: 0023
CS: 001B
```

使用小程序 *SegId.exe*(或使用计算工具)使寄存器值逻辑右移 3 位。得到段选择器值：

```
C:\>SegId $0038 $0023 $001B
$0038=> 7
$0023=> 4
$001B=> 3
```

程序 *SegId.exe* 的源代码如下：

```
{$APPTYPE CONSOLE}
program SegId;

var
  i, v, tag: Integer;

function id(s:string) : string;
begin
  result := '????';
```

① 这需要求先了解一些关于 Win32 的内存地址映射机制和 GDT(Global Descriptor Table，全局描述表)的知识。

```
  val(s, v, tag);
  if tag=0 then str(v shr 3, result);
end;

begin
  for i:=1 to ParamCount() do
    writeln(ParamStr(i):5, '=> ', id(ParamStr(i)));
end.
```

接下来要从 GDT 中按段选择器索引来查找对应的虚地址。通常这要使用 SoftIce 或者 WinDBG。这里用一个名为 DumpXDT 的小工具来列出 GDT：

```
C:\>DumpXDT
GDTbase=8003F000  Limit=03FF
0000  0  NP  Reserved    00000000 00000000
0008  0  P   Code32      00000000:FFFFFFFF  RE
0010  0  P   Data32      00000000:FFFFFFFF  RW
0018  3  P   Code32      00000000:FFFFFFFF  RE
0020  3  P   Data32      00000000:FFFFFFFF  RW
0028  0  P   TSS32       80042000:000020AB  A
0030  0  P   Data3       FFDFF000:00001FFF  RW
0038  3  P   Data32      7FFDE000:00000FFF  RW
0040  3  P   Data16      00000400:0000FFFF  RW
// ...
```

在所列出的 GDT 中，第零段：

```
0000  0  NP  Reserved  00000000 00000000
```

的类型为“Reserved”(保留)。第一段：

```
0008  0  P   Code32    00000000:FFFFFFFF  RE
```

的类型是“Code32”，属性为“RE”，表明这是一个可读(R)、可执行(E)的代码段(Code32)。同理，FS 段选择器的索引值为 7，表明它指向第七段：

```
0038  3  P   Data32    7FFDE000:00000FFF  RW
```

这段是一个可读(R)写(W)的数据段。段的基地址(虚拟地址)是$7FFDE000，段的大小为$0FFF字节(4096)字节。

FS:[0]则表示以虚地址$7FFDE000 为基地址，偏移为 0 的地址。现在，将 Delphi IDE 的 CPU 窗口的数据区定位到$7FFDE000，就找到 FS:[0]所指向的内存了（图 9.4）。

可以看到在$7FFDE000 这个地址上(即 FS:[0])正好指向栈上的$12FE24 这个地址。而在

这个地址中是一个异常帧：

```
0012FE2C : 0012FE40       ==> push ebp
0012FE28 : 00443E6D       ==> push @HandleAnyExcpet(,or other Handle)
0012FE24 : 0012FE5C       ==> push dword ptr fs:[eax]
```



图 9.4 FS:[0]指向的内存

现在可以回到第 8 章第 4 节“从操作系统的角度理解编译器行为”中，来理解这一句叙述了：FS 段寄存器指向一个 TIB(Thread Information Block)结构。而这个结构定义如下：

```
_NT_TIB = record
   // 00h Head of exception record list
   ExceptionList: PEXCEPTION_REGISTRATION_RECORD;

// ...
```

最后花一点时间来理解 CS 与 DS。

CS 与 DS 寄存器指向不同的段选择器索引。在上面的例子中，它们分别指向 GDT 中的第三、四段：

```
0018  3  P  Code32    00000000:FFFFFFFF  RE
0020  3  P  Data32    00000000:FFFFFFFF  RW
```

可以看出，这两个段的起始虚地址和大小都是一致的。不过它们的属性并不相同，CS 指向的是代码段，具有可执行权限(E)，但是只读(R)；而 DS 指向数据段，可读写(RW)，但没有执行权限。它们的地址空间大小都是 4G($FFFFFFFF)。

非常明显的一个结论是：在 CPU 窗口中，可以在数据区中使用相同的虚地址来查看代码段

中的二进制数据，其根本原因在于 CS 与 DS 所指向的段基址都是$00000000。相类同的道理，FS:[0]与 DS:[7FFDE000]所指向的也是相同的内存虚地址。

GDT 已经揭示了进程地址空间的绝大多数特性，至于应用程序可以存取 4G 空间中的哪些部分，则是操作系统另外的限制，与这里讨论的段寄存器就没有直接关系了。

9.6　线程类(TThread)

大多数开发人员不会使用 API：CreateThread()或者内核例程 BeginThread()来创建新线程。作为 Delphi 推荐的方法，基于 TThread 派生一个新类，是一个不错的主意。

线程类 TThread 位于 *Classes.pas* 单元中，它封装了操作系统中的线程对象[①]。由于每个 TThread 的实例对应于一个线程，这意味着它至少需要拥有以下属性：

```
// code from Classes.pas
TThread = class
private
  // ...
protected
  // ...
public
  // 构造函数与析构函数
  constructor Create(CreateSuspended: Boolean);
  destructor Destroy; override;

  // 线程的基本操作(方法)
  procedure Resume;
  procedure Suspend;

  // 线程的基本属性
  property Handle: THandle read FHandle;
  property Priority: TThreadPriority read GetPriority write SetPriority;
  property Suspended: Boolean read FSuspended write SetSuspended;
  property ThreadID: THandle read FThreadID;
  // ...
end;
```

① 操作系统将线程、进程等都称为对象，这与 OOP 中的“类的实例-对象”这一概念完全不同。

对于线程的几个基本操作的 API：

```
// 设置与获取线程的优先级
function SetThreadPriority(hThread: HANDLE;
  nPriority: Integer): BOOL; stdcall;
function GetThreadPriority(hThread: HANDLE): Integer; stdcall;

// 挂起一个线程(使代码暂停执行)
function SuspendThread(hThread: HANDLE): DWORD; stdcall;

// 恢复一个被挂起的线程(使代码在上一次暂停处继续执行)
function ResumeThread(hThread: HANDLE): DWORD; stdcall;
```

来说，Delphi 将它们封装成对象方法或属性的重要原因之一，是使得线程操作过程中的出错信息可以被转化为 Delphi 的异常对象，从而使用户代码可以更加方便地处理它们。这种转换的方法非常简单：

```
// code from Classes.pas

// 如果出错，则通过出错代码获取出错信息，然后创建异常对象并触发它
procedure TThread.CheckThreadError(ErrCode: Integer);
begin
  if ErrCode <> 0 then
    raise EThread.CreateResFmt(@SThreadError,
      [SysErrorMessage(ErrCode), ErrCode]);
end;

procedure TThread.CheckThreadError(Success: Boolean);
begin
  if not Success then
    CheckThreadError(GetLastError);
end;

function TThread.GetPriority: TThreadPriority;
// ...
  P := GetThreadPriority(FHandle);
  CheckThreadError(P <> THREAD_PRIORITY_ERROR_RETURN);
// ...
end;

procedure TThread.SetPriority(Value: TThreadPriority);
begin
  CheckThreadError(SetThreadPriority(FHandle, Priorities[Value]));
end;
```

```
procedure TThread.Suspend;
// ...
   CheckThreadError(Integer(SuspendThread(FHandle)) >= 0);
// ...
end;

procedure TThread.Resume;
//...
  SuspendCount := ResumeThread(FHandle);
  CheckThreadError(SuspendCount >= 0);
//...
end;
```

这种将出错代码封装成异常的技巧同样可以用在用户代码中。例如设计一个封装了安全 API 调用机制的类：

```
resourcestring
  SApiCallerError = 'Api Call Error: %s (%d)';

type
  EApiCaller = class(Exception);
  TApiCaller = class
  protected
    procedure CheckThreadError(ErrCode: Integer); overload;
    procedure CheckThreadError(Success: Boolean); overload;
  public
    function GetDC(hwnd: HWND) : HDC;
    // ... 更多的API
  end;

// ...

procedure TApiCaller.CheckThreadError(ErrCode: Integer);
begin
  if ErrCode <> 0 then
    raise EApiCaller.CreateResFmt(@SApiCallerError,
      [SysErrorMessage(ErrCode), ErrCode]);
end;

procedure TApiCaller.CheckThreadError(Success: Boolean);
begin
  if not Success then
    CheckThreadError(GetLastError);
end;
```

Delphi 源代码分析

```
function TApiCaller.GetDC(hwnd: HWND) : HDC;
begin
  Result := windows.GetDC(hwnd);
  CheckThreadError(Result <> 0);
end;
```

除此之外，线程类中最重要的几个方法是：

```
// defined in Classes.pas, Class TThread

// ...
  protected
    procedure Execute; virtual; abstract;
    procedure Synchronize(Method: TThreadMethod); overload;
  public
    procedure Terminate;
    function WaitFor: LongWord;
// ...
```

9.6.1 线程执行体(Execute)

线程对象将执行的用户代码设计为虚方法 **Execute()**。这意味着用户必须在一个派生的新线程类中重写 **Execute** 方法。而这个方法最终将被例程 **ThreadProc()** 调用到：

```
// code from Classes.pas
Function ThreadProc(Thread: TThread): Integer;
// ...
    if not Thread.Terminated then
    try
      Thread.Execute;
// ...
```

线程的构造函数中将调用内核例程 **BeginThread()**，以使得在每个线程被创建时，线程入口代码总是指向 **ThreadProc()**。这样也就完成了线程执行体(代码)的封装：

```
// code from Classes.pas
constructor TThread.Create(CreateSuspended: Boolean);
// ...
  FHandle := BeginThread(nil, 0, @ThreadProc, Pointer(Self),
    CREATE_SUSPENDED, FThreadID);
// ...
```

但通过线程类(及其派生类)创建的线程，缺省总是被挂起的——可以看到：在构造函数中，调用 **BeginThread()** 时总是传入 **CREATE_SUSPENDED** 参数。

那么，线程执行体到什么时候才会被执行呢？

TThread 类定义中，有两个私有字段是与线程的初始状态有关的：

```
// defined in Classes.pas
 TThread = class
 private
   FCreateSuspended: Boolean;
   FSuspended: Boolean;
// ...
 public
   property Suspended: Boolean read FSuspended write SetSuspended;
```

FCreateSuspended 保存 TThread.Create()传入的 CreateSuspended 参数值，指明用户代码在线程创建时挂起。而 FSuspended 则指明线程当前的挂起状态。在 Create()方法中：

```
// code from Classes.pas
constructor TThread.Create(CreateSuspended: Boolean);
begin
 // ...
 FSuspended := CreateSuspended;
 FCreateSuspended := CreateSuspended;
```

尽管线程总是以挂起状态创建，但初始时 FSuspended 中仍直接保存了来自 CreateSuspended 参数的值。这是一个很不起眼的 BUG。因为它将导致在 Create()方法调用后、AfterConstruction()调用前无法有效地判定线程状态。如果线程创建时指定参数 CreateSuspended 为 False，则线程会在 AfterConstruction()中被恢复执行：

```
// code from Classes.pas
procedure TThread.AfterConstruction;
begin
  if not FCreateSuspended then
    Resume;
end;
```

但无论如何，在上面这行代码被执行前，TThread 类的 FSuspended 字段和 Suspended 属性与线程的实际状态是不一致的。如果开发人员试图在子类中用这样的代码来覆盖 TThread.AfterConstruction()，如下所示：

```
procedure TSubThread.AfterConstruction;
begin
```

```
  if FSuspended then
  begin
    // ...
  end;

  inherited;
end;
```

那么开发人员将面临一个不正确的 FSuspended 状态属性。因此，在 TThread 类的实现中，更为安全的代码应当是这样的：

```
// code from Classes.pas
constructor TThread.Create(CreateSuspended: Boolean);
begin
  FSuspended := True;
  FCreateSuspended := CreateSuspended;
  // ...
end;

procedure TThread.AfterConstruction;
begin
  if not FCreateSuspended then
  begin
    FSuspended := False;
    Resume;
  end;
end;
```

通过覆盖 AfterConstruction()方法，Delphi 内核将“对象创建”与“线程执行”两种行为分隔开来。简单地说，在设计子类时：

- 要改变线程对象的创建方法，例如添加和初始化对象的字段，或者改变传入的参数，那么应该覆盖 Create()方法；
- 如果要组合并识别一系列字段，使用更复杂的线程初始执行条件(是否被挂起)，则应当覆盖 AfterConstruction()方法。

9.6.2 线程与主线程的同步(Synchronize)

从本意上来讲，Synchronize()方法被设计出来的目的，是用来使新线程与进程的主线程同步。但这个很明确的设计目的，被绝大多数的资料描述成：同步新线程与主线程之间对界面(GUI)的操作。

可以用 Synchronize() 方法来同步任何主线程与其他线程之间的操作，这与 VCL 无关，与窗体无关。主线程如何响应这个同步执行的请求是不重要的：用窗体来响应，抑或是对象或者其他执行机制来响应，是“(发出同步请求)以后”的事——这样的细节被 Delphi 内核的设计人员放在了 *Classes.pas* 单元之外。

■ 线程同步方法：Synchronize()

在 *Classes.pas* 中，线程同步方法事实上只负责向全局变量 SyncList 添加一个同步执行过程记录(TSyncProc)指针，该记录声明如下：

```
// code from Classes.pas
type
  TSyncProc = record
    SyncRec: PSynchronizeRecord;
    Signal: THandle;
  end;
  PSyncProc = ^TSyncProc;

var
  SyncList: TList = nil;
  ThreadLock: TRTLCriticalSection;
  ThreadCount: Integer;
```

根据不同的应用场合和对象层次结构的设计需要，在 TThread 类中定义了四个线程同步方法，代码如下：

```
// code from Classes.pas
 TThread = class
 private
   class procedure Synchronize(ASyncRec: PSynchronizeRecord); overload;

 protected
   procedure Synchronize(Method: TThreadMethod); overload;

 public
   class procedure Synchronize(AThread: TThread;
     AMethod: TThreadMethod); overload;
   class procedure StaticSynchronize(AThread: TThread;
     AMethod: TThreadMethod);
// ...
```

所有的线程同步方法最终都将调用私有类方法：

```
// defined in Classes.pas, Class TThread
  class procedure Synchronize(ASyncRec: PSynchronizeRecord); overload;
```

这事实上也表明：可以在不创建 TThread 对象实例的情况下，向系统的 SyncList 中添加一个同步记录。但由于这是一个私有方法，因此只可能通过另外两个类方法来达到这样的目的：

```
// defined in Classes.pas, Class TThread
  class procedure Synchronize(AThread: TThread;
    AMethod: TThreadMethod); overload;
  class procedure StaticSynchronize(AThread: TThread;
    AMethod: TThreadMethod);
```

■ 当前线程调用 Synchronize()来准备同步记录

除了 StaticSynchronize()这个名字更确切地表明了设计它的目的之外，这两个类方法的实现并没有明显的不同。因为 StaticSynchronize()只是简单地调用 Synchronize()：

```
// code from Classes.pas
class procedure TThread.Synchronize(AThread: TThread;
  AMethod: TThreadMethod);
var
  SyncRec: TSynchronizeRecord;
begin
  if AThread <> nil then
    AThread.Synchronize(AMethod)
  else
  begin
    SyncRec.FThread := nil;
    SyncRec.FSynchronizeException := nil;
    SyncRec.FMethod := AMethod;
    TThread.Synchronize(@SyncRec);
  end;
end;
```

这里的 SyncRec 是一个在栈上分配的 TSynchronizeRecord 记录。那么，如果要保证被同步方法 AMethod 在执行时有正确的栈结构，则：

- AMethod 在一个线程中被执行，在开始执行前该线程被挂起；或，
- AMethod 是主线程中的一个方法，可在主线程中直接执行而不必同步。

由此可见，方法 AMethod 能否被正确执行并不取决于 AThread 是否有效地指向一个 TThread 对象实例，而是取决于执行期该线程的状态。所以 AThread 这个参数可以传入 nil。

也因此类方法 Synchronize()也可以将为 nil 值的 AThread 封装到 SyncRec 记录中，而不必担心空的对象实例指针对后续代码造成负面影响。接下来调用私有的类方法：

```
// code from Classes.pas
class procedure TThread.Synchronize(ASyncRec: PSynchronizeRecord);
var
  SyncProc: TSyncProc;
begin
  if GetCurrentThreadID = MainThreadID then
    ASyncRec.FMethod
  else
    // ...
```

如前所述，类方法 TThread.Synchronize()果然检测当前线程是否是主线程，如果是，则 ASyncRec.FMethod 将被直接调用。这与在单线程模式下直接调用一个方法并没有不同，也就是说，(如果当前线程是主线程)下面两段代码的效果是一致的：

```
// 代码段一
procedure TForm1.Button1Click(Sender: TObject);
begin
  TThread.StaticSynchronize(nil, Button2.Click);
end;

// 代码段二
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button2.Click;
end;
```

■ 当前线程调用 WakeMainThread()，随后挂起线程等待

如果当前线程不是主线程，那么调用类方法 TThread.Synchronize()时，执行的流程与其他(有线程对象实例)会是这样：

```
// code from Classes.pas
class procedure TThread.Synchronize(ASyncRec: PSynchronizeRecord);
// ...
    EnterCriticalSection(ThreadLock);
    try
      if SyncList = nil then
        SyncList := TList.Create;
      SyncProc.SyncRec := ASyncRec;
      SyncList.Add(@SyncProc);
```

```
        SignalSyncEvent;
        if Assigned(WakeMainThread) then
          WakeMainThread(SyncProc.SyncRec.FThread);
        LeaveCriticalSection(ThreadLock);
        try
          WaitForSingleObject(SyncProc.Signal, INFINITE);
// ...
```

ASysRec 记录被封装到另一个线程局部变量 SyncProc 中，然后 SyncProc 记录被添加到 SyncList 列表中。

接下来调用例程 SignalSyncEvent()置线程同步事件标志量：

```
// code from Classes.pas
procedure SignalSyncEvent;
begin
  SetEvent(SyncEvent);
end;
```

SyncEvent 在 *Classes.pas* 单元初始化时通过调用例程 InitThreadSynchronization()来创建和初始化。这使得引用了 *Classes.pas* 单元的应用程序默认都具有多线程下执行和同步的能力——与大多数人设想的不同：这种特性是不需要通过创建 TThread 的对象实例就可以获得的。

随后执行到的代码是：

```
      if Assigned(WakeMainThread) then
        WakeMainThread(SyncProc.SyncRec.FThread);
```

这两行代码用于通告主线程有一个同步方法需要执行。随后，当前线程挂起并等待标志量 SyncProc.Signal 信号：

```
        WaitForSingleObject(SyncProc.Signal, INFINITE);
```

当前线程的执行流程将因为等待信号而暂停。因此在类方法 Synchronize()中，在栈上分配的两个记录也因为过程并没有执行结束(没有执行到过程的 ret 指令)，而暂时不被释放：

```
// code from Classes.pas
class procedure TThread.Synchronize(AThread: TThread;
  AMethod: TThreadMethod);
var
  SyncRec: TSynchronizeRecord;
  // ...
```

```
class procedure TThread.Synchronize(ASyncRec: PSynchronizeRecord);
var
  SyncProc: TSyncProc;
// ...
```

另一种复杂的情况是：当前线程既不是主线程，也不是用 TThread 创建的线程。例如：

```
function ThreadFunc(Parameter: Pointer): Integer; stdcall;
begin
  // 当前线程是使用API直接创建的
  TThread.StaticSynchronize(nil, Button2.Click);
end;

var
  ThreadID : Cardinal;
  ThreadHandle : Cardinal;
procedure ThreadExecuteTest;
begin
  ThreadHandle := CreateThread(nil, 0, @ThreadFunc, nil, 0, ThreadID);
end.
```

在这个例子中，系统的确处于多线程状态下，但是向 StaticSynchronize 传入的 AThread 值为 nil，因为当前线程是使用 API 创建的。

那么在调用 WakeMainThread()时传入的 SyncRec.FThread 值也为 nil。这时，一切又会如何呢？

从在 *Forms.pas* 单元中 TApplication 的实现代码里可以发现，WakeMainThread 根本就不使用这个线程对象的引用。

```
// code form Forms.pas
procedure TApplication.WakeMainThread(Sender: TObject);
begin
  PostMessage(Handle, WM_NULL, 0, 0);
end;

procedure TApplication.HookSynchronizeWakeup;
begin
  Classes.WakeMainThread := WakeMainThread;
end;

constructor TApplication.Create(AOwner: TComponent);
begin
  // ...
  HookSynchronizeWakeup;
```

```
end;
```

在使用 VCL 的多线程应用中，新线程同步方法向主线程传入的 FThread 并没有被使用到，主线程只是向窗体消息队列中塞入了一个 WM_NULL 消息。而且同步方法指针 SyncProc.SyncRec.FMethod 也并没有传入主线程。

■ 主线程回调 CheckSynchronize()例程

进一步的问题是：新线程的同步方法是怎么被执行到的呢？

主线程的窗体消息响应例程接到 WM_NULL 消息时，将会主动调用 Classes.CheckSynchronize()例程：

```
// Code from Forms.pas
procedure TApplication.WndProc(var Message: TMessage);
// ...
  with Message do
    case Msg of
      // ...
      WM_NULL:
        CheckSynchronize;
    else
// ...
end;
```

现在执行流程又回到了 *Classes.pas* 单元。在进一步地分析之前，应当注意的几点事实是：

- (到目前，)主线程并不执行同步方法也不响应它的执行，主线程只是在自己空闲时调用一下 CheckSynchronize()例程即可。
- 主线程并不关心线程同步代码的执行细节，也不关心被同步线程的对象实例或者方法指针。
- (在新线程中，)被同步的线程方法并不知道主线程是执行在控制台还是 GUI 模式下，也没有相关的代码来辨识主线程。这意味着一些资料中关于“由于需要同步的过程被放在主线程中执行，所以在一些没有消息循环的应用中（如 Console 或 DLL）无法使用”这样的认识是错误的。

事实上，按原理来说，需要同步的过程应当在自己的线程中执行。所谓同步，只是避免与主线程同时执行，而并不是要在主线程中执行这个过程。然而在 TThread.Synchronize()中传入的同步方法，却会在主线程中执行。因此 CheckSynchronize()例程中会检测当前线程是否是主线程，以避免在其他线程中调用这个函数：

```
// code from Classes.pas
function CheckSynchronize(Timeout: Integer = 0): Boolean;
var
  SyncProc: PSyncProc;
  LocalSyncList: TList;
begin
  if GetCurrentThreadID <> MainThreadID then
    raise EThread.CreateResFmt(@SCheckSynchronizeError,
      [GetCurrentThreadID]);
// ...
```

接下来，主线程将用一个空的LocalSyncList对象去替换出系统正在使用的SyncList：

```
// code from Classes.pas, function CheckSynchronize()
// ...
  LocalSyncList := nil;
  // ...
    Integer(LocalSyncList) := InterlockedExchange(Integer(SyncList),
      Integer(LocalSyncList));
```

最后一行代码的操作相当于：

```
var
  tempInt : Integer;

// ...
  tempInt := Integer(SyncList);
  Integer(SyncList) := Integer(LocalSyncList);
  Integer(LocalSyncList) := tempInt;
```

使用API：InterlockedExchange()这个原子操作函数的目的，是要避免LocalSyncList与SyncList这两个变量互相交换的执行过程被其他线程打断。这是多线程同步的问题，在后面的小节中将会继续讲述。

最后CheckSynchronize()使用一个循环来调用LocalSyncList中的全部被同步的方法：

```
// code from Classes.pas, function CheckSynchronize()
// ...
      while LocalSyncList.Count > 0 do
      begin
        SyncProc := LocalSyncList[0];
        LocalSyncList.Delete(0);
        // ...
          SyncProc.SyncRec.FMethod;
```

最后调用 API：SetEvent()来置 SyncProc.signal 并释放 LocalSyncList。等待 SyncProc.signal 消息量发生变化的线程在随后也会得到信号[①]，然后从等待状态恢复并继续执行：

```
// code from Classes.pas, function CheckSynchronize()
// ...
        SetEvent(SyncProc.signal);
// ...
    LocalSyncList.Free;
```

应该注意到这里没有任何地方需要访问 SyncRec 的另外一个域：TSynchronize.FThread。

■ 被同步方法传入的 Self 对象

大多数情况下，一个被同步方法总是在 TThread 类中被定义的。例如 Delphi 官方帮助文档中的示例代码：

```
procedure TMyThread.PushTheButton;
begin
  Button1.Click;
end;

procedure TMyThread.Execute;
begin
  // ...
  Synchronize(PushTheButton);
  // ...
end;
```

那么，在 PushTheButton()中访问 Self 对象应当是 TMyThread 的一个实例。但这并不表明这里的 Self 一定是一个 TThread 的子类对象，或者是有效的对象实例。下面这样的代码事实上是允许的：

```
procedure TForm1.PushTheButton;
begin
  Button1.Click;
end;
```

① 如何唤醒处于等待状态的线程，是操作系统的线程调度机制的事。API：SetEvent()只负责设置信号量的状态。也就是说，等待线程并不是在 SetEvent()调用结束后就立即开始执行的。

```
procedure TMyThread.Execute;
begin
  // ...
  Synchronize(form1.PushTheButton);
  // ...
end;
```

这个例子中，PushTheButton()中的 Self 是 TForm1 的一个实例。而下面的代码中，PushButton 最终将会作为方法被调用，且它的 Self 值将是 nil[①]：

```
procedure PushButton;
begin
  Form1.Button1.Click;
end;

// 尝试使用线程同步方法来执行普通过程PushButton()
procedure ExecutePushButton;
var
  FMethod : TMethod;
begin
  FMethod.data := nil;
  FMethod.code := PushButton;
  TThread.Synchronize(nil, FMethod);
end;

//  在一个线程中执行ExecutePushButton()
procedure ThreadExecutePushButton;
var
  ThreadID : Cardinal;
  ThreadHandle : Cardinal;
begin
  ThreadHandle := CreateThread(nil, 0, @ ExecutePushButton, nil, 0,
    ThreadID);
end;
```

ThreadExecutePushButton()例程的代码涉及到如下三个问题：

- 一个无效的 AThread 值：nil；
- 一个无效的 Self 值：nil；
- 当前线程可能是主线程。

① 这里使用了手工构建对象方法的技巧。如果想在 PushButton()过程中访问这个 Self 值(尽管它是 nil)，那么应当声明一个入口参数。从 OOP 设计的角度上来说，这非常不可取。

TThread 类的设计中避开了这些问题。因为在整个执行过程中，Self 值和 AThread 并不被直接使用，而识别线程对象有效性的任务，事实上是交给了用户代码：被同步的过程(或方法)。此外，WakeMainThread()中对主线程的识别，也避免了出现“主线程等待主线程执行 CheckSynchronize()”这样的逻辑错误。

向被同步的方法传入 Self 值的行为，其实与 Delphi 中实现 OOP 的机制有关。在 Class.CheckSynchronize()中只是简单地调用：

```
SyncProc.SyncRec.FMethod;
```

来执行被同步的方法。而 FMethod 中需要用到的那个 Self 值，是编译器在发现 FMethod 是一个 TMethod 类型的对象方法时，在调用例程前面插入一段代码，从 TMethod(FMethod).data 中取出 Self 值并放到 EAX 中去。

AThread 对 TThread 类的同步方法 Synchronize()的设计本身来说并没有意义，除非用户愿意设计第三方的 WakeMainThread()例程来响应线程同步请求。

9.6.3 控制台应用的多线程同步(第三方的 WakeMainThread 例程)

某些情况下的控制台应用程序设计，可能会要求控制台只是信息的输出界面，但同时会有多个线程向它输出。例如一个检测硬件设备状况的控制台应用框架可能会是这样：

```
program MutilThreadConsole;

{$APPTYPE CONSOLE}

uses
  Windows, Messages, SysUtils, Classes;

var
  ThreadArr : array of record
    Handle, ID : Cardinal;
  end;

var
  Str : String;
  i : Integer;
  iMax : Integer = 500;
  step : Integer = 1;

function OutThreadFunc(Parameter: Pointer): Integer; stdcall;
begin
```

```
  while true do
  begin
    writeln('need your input: ');
    sleep(step);
  end;
end;

begin
  // 创建足够多的输出线程
  SetLength(ThreadArr, iMax);
  for i:=0 to iMax-1 do
    ThreadArr[i].Handle := CreateThread(nil, 0, @OutThreadFunc, nil,
      0, ThreadArr[i].ID);

  // 等待输入或其他界面响应
  repeat
    readln(Str);
    // do any...
  until Str <> '';

  // 释放线程
  for i:=0 to iMax-1 do
    CloseHandle(ThreadArr[i].Handle);
end.
```

这个例子中可能有 iMax 个输出线程在使用控制台输出。如果 iMax 值足够大，且 step 值足够小(或随机)，那么就很容易导致控制台的争用(冲突)。例如出现图 9.5 所示的这种情况(iMax=300,step=1)：



图 9.5　控制台的争用

这时就应当在多个线程输出中同步写控制台的函数 Writeln()。

Delphi 编译的窗体应用程序可以使用同步方法 Synchronize()，是因为 Forms 单元中使用了 TApplication.WakeMainThread()来接管 Class.WakeMainThread()。因此如果需要在控制台中使用类同的机制，则程序员应该自己编写一个第三方的 WakeMainThread()例程。

接管该例程的核心问题在于“何时调用 CheckSynchronize()例程”。

在控制台应用程序中，通常要求主程序针对控制台上的“输入 / 输出”来控制程序的流程。例如在按下键盘上的数字键“1”就进入菜单功能项 1。这是一个惯常的设计，但这也导致了用户界面(UI)上的消息阻塞。简单地在流程中加入对 CheckSynchronize()的调用，要么是流程很快地结束，要么是 UI 消息阻塞导致其他线程被同步的方法不能实时(或者说平滑)地被执行。

可以给控制台加入 Timer 线程来解决消息实时的问题，也就是说，使真正同步执行的线程不是主线程而是 Timer 线程。

当然还应当使其他的线程通过 TThread.Synchronize()来调用 Writeln()方法。这样，在主线程中用户界面上的等待就不会影响执行流程了。下面是一个简单的实现框架：

```
// 使用时间和控制台消息队列来实现同步的单元
unit ConsoleTimerWaker

interface
uses
  Windows, Messages, Classes;

var
  MainThreadID : Cardinal;      // 用于替代System.MainThreadID;

implementation

procedure WakeConsole(Sender: TObject);
begin
  PostThreadMessage(MainThreadID, WM_TIMER, 0, 0);
end;

var
  WatinThreadHandle : Cardinal; // 处理消息循环的线程句柄
  WakeMainThread : TMethod = (Code:@WakeConsole; Data:Nil);

procedure TimerCallBack(hWnd: HWND; Msg, idEvent: UINT;
  dwTime: DWORD); stdcall;
begin
  Classes.CheckSynchronize;
end;
```

```
function WatchThreadFunc(Parameter: Pointer): Integer; stdcall;
var
  TimerHandle : THandle;
  Msg : TMsg;
begin
  Classes.WakeMainThread := TNotifyEvent(WakeMainThread);
  TimerHandle := SetTimer(0, 0, 100, @TimerCallBack);
  if TimerHandle = 0 then
    writeln('could not create timer')
  else
  begin
    // 实现消息循环处理(相当于WndProc)
    repeat
      GetMessage(Msg, 0, 0, 0);
      DispatchMessage(Msg);
    until (Msg.Message = WM_QUIT);
    KillTimer(0, TimerHandle);
  end;
end;

initialization
  MainThreadID := System.MainThreadID;
  WatchThreadHandle := CreateThread(nil, 0, @WatchThreadFunc,
    nil, 0, System.MainThreadID);

finalization
  CloseHandle(WatchThreadHandle);
  System.MainThreadID := MainThreadID;

end.


// 使用示例(部分代码参见上例)
program ConsoleWakeDemo;

{$APPTYPE CONSOLE}

uses
  ConsoleTimerWake in 'ConsoleTimerWake.pas',
  Windows, Classes;

var
  // ...

type
  TSyncObj = class
  private
```

```
    InfoString : String;
  public
    procedure SyncOutputProc;
  end;

procedure TSyncObj.SyncOutputProc;
begin
  Writeln(InfoString);
end;

function OutThreadFunc(Parameter: TSyncObj): Integer; stdcall;
begin
  while true do
  begin
    Parameter.InfoString := 'need your input: ';
    TThread.Synchronize(nil, Parameter.SyncOutputProc);
    sleep(step);
  end;
end;

// 开始(控制台)主线程代码
begin
  SetLength(ThreadArr, iMax);
  for i:=0 to iMax-1 do
  begin
    ThreadArr[i].Handle := CreateThread(nil, 0, @OutThreadFunc,
      TSyncObj.Create, 0, ThreadArr[i].ID);
  end;

  // ...
end.
```

该示例程序的运行结果如图 9.6(很明显，输出同步的问题被解决了)。



图 9.6 例程运行结果

9.6.4　线程中止和结束(Terminate)

线程的结束与中止不是同一个概念。通常把这下列方法导致的线程终结归为“结束”：

- 线程执行体(过程)执行完毕并正常返回(RET 指令)；
- 线程内的代码调用 API：ExitThread()；

而下列方法导致的线程终结被归为“中止”：

- 调用 API：TerminateThread()；
- 调用 API：ExitProcess()或 TerminateProcess()使进程结束，同时线程被强制中止。

无论是中止或者结束，(先假定操作能够成功)其结果都是线程生存期的终结。因此一旦这种情况发生(无论是线程自己抑或其他线程调用 API)，被中止线程都不会再得到代码执行的机会。这与线程间相互等待(死锁)的状态不相同：死锁是 A、B 两线程的生存期都没有结束。

后续的问题仍然极其复杂。

问题复杂的真正的原因在于：操作系统没有一种机制，能够发送一个“线程终结”的通知消息到线程，并且使线程响应它。因为操作系统无法判定 API：TerminateThread()操作的目标线程是已经被死锁，还是在正常执行中被挂起。这样，如果 A、B 线程通过一个进程间标志来识别相互间的运行状态，假定 A 线程被强制中止，那么 B 线程可能因为一直在等待而挂起。

通常线程中止的请求总是能被操作系统很快地完成。因此如果一个线程总是不能结束，那么可以尝试中止它——尽管所有与操作系统相关的书都提示开发人员不要这样做，但这的确是最后的一根稻草。

如果说还有任何其他的途径的话，那就只剩下中止进程了。而这仅仅是为了保障操作系统平台仍然能够运行而已。例如可以先中止一个系统服务，然后再重新启动它(创建新的进程)，这样可能避免重启机器。

使用 API：ExitThread()结束线程与使用 TerminateThread()并没有本质上的不同。但前者是当前线程主动结束自己，而后者是强制中止目标线程。尽管操作系统对两个 API 的处理大致相同，但从设计的角度来看，却是有着较大差异的。

■ *偷换概念：Delphi 中的线程中止*

Delphi 的 TThread 中的 Terminate()方法偷换了操作系统中线程中止的概念。这使得

Delphi 中的 TThread 对象在终止或结束时都显得与上面讲述过的内容不同。这也使得从其他语言(如汇编和 C)转到 Delphi 中开发的程序员难以适应。所以大多数情况下，这些开发人员还是习惯用 API 来创建线程，并自己管理线程句柄和线程 ID。

在 Delphi 的 TThread 类的使用中，要求开发人员在 Execute()方法的代码中自行检测 TThread.Terminated 标志并决定是否退出 Execute()方法。这事实上就与“线程中止”这个概念在操作系统中的原意相悖。在线程对象的(公共)执行过程 ThreadProc()中：

```
// code from Classes.pas
function ThreadProc(Thread: TThread): Integer;
begin
// ...
    if not Thread.Terminated then
    try
      Thread.Execute;
//...
  finally
    FreeThread := Thread.FFreeOnTerminate;
    Result := Thread.FReturnValue;
    Thread.DoTerminate;
    if FreeThread then Thread.Free;
    EndThread(Result);
  end;
end;
```

ThreadProc()保证线程对象的 Execute()方法在一个 try..finally 块中执行，直到 Execute()方法退出或者异常，执行流程进入到 finally..end 块。然后调用线程对象的 OnTerminate 事件的响应例程，如果 FreeOnTerminate 属性设置为 True 的话，此后线程对象将会被自动释放。

FFreeOnTerminate 和 FReturnValue 两个字段的值在调用 DoTerminate()方法前被取出，使得用户代码在响应 OnTerminate 事件时对这两个值的修改变得没有意义。这样设计具有一定的合理性：

- 对象何时被释放应当是在它创建时就能预测的，一个在运行过程中会动态确定自己何时被 Free 的对象显然并不安全。因此 FFreeOnTerminate 应当在子类的构造函数 Create()中设定[①]。
- 线程的执行状态应该在执行过程中被监测，因此执行期返回的(状态或结果)值不应该被 OnTerminate 响应例程修改。事实上，从设计的角度上来讲，OnTerminate 只

① 如果不愿意覆盖 Create()方法，那么也可以在 Execute()的开始处设定，但这不是好的设计风格。

决定线程中止时应当做什么，而不应决定线程如何中止或者如何返回。

举例来说，FreeOnTerminate 置为 False 值的线程对象在 Execute()执行结束后不会被释放。通常可以把这个 TThread 的子类对象缓存到一个列表、队列或者栈中[①]。而“线程对象是否被缓存”，是在设计期就可以决定的事，而不需要等到运行期在 OnTerminate 事件响应例程中去动态调整——如果线程已经塞入队列或栈，那么重设 FreeOnTerminate 值将导致 ThreadProc()自动释放该对象，而队列或栈中也将出现一个无效的对象指针。

可见，FTerminated 字段并不决定线程对象的生死。在 TThread 中，线程对象的生存周期决定于：

- FTerminated 是否为 True，这决定了 Execute 是否执行；
- Execute()是否执行完毕，或者是否中途退出(用户代码检测 FTerminated 值)；
- FreeOnTerminate 是否为 True。

TThread.Terminate 方法仅将 FTerminated 置为 True，因此它并没有与 API：TerminateThread()相同的效果。线程对象(TThread)中的线程是否能结束或中止，仍旧取决于上面的三个条件。

```
// code from Classes.pas
procedure TThread.Terminate;
begin
  FTerminated := True;
end;
```

Delphi 并不赞成开发人员调用不安全的线程中止例程 API：TerminateThread()。这是 TThread 类中使用 Terminate 方法来替代这个 API 的惟一原因。尽管使用 Terminate 方法显得很繁琐，但它的确要安全得多。惟一的问题是，开发人员必须在 Execute()中不断地复核 Terminated 属性。

和 API：TerminateThread()一样，TThread.Terminate 方法可以从线程之外（其他线程中）调用。不同的是，TThread.Terminate 调用后并不会立即中止线程，线程也未必结束生存周期。

■ 线程生存周期的结束：ExitThread()

通过 FreeOnTerminate 属性，Delphi 把线程作为“TThread 对象”时的析构与作为“操

① 这是线程队列和线程池的概念。在需要大量重用线程时，缓存对象实例可以明显地改善性能。

作系统线程(执行体)”时的结束两个概念分离开来：

```
// code from Classes.pas
function ThreadProc(Thread: TThread): Integer;
begin
// ...
    if not Thread.Terminated then
    try
      Thread.Execute;
//...
  finally
    FreeThread := Thread.FFreeOnTerminate;
// ...
    if FreeThread then Thread.Free;
    EndThread(Result);
  end;
end;
```

可见，操作系统所理解的线程结束发生于“线程执行体”执行结束之后：一旦线程代码执行完，那么线程总是会被结束的。内核例程 System.EndThread()将调用 API：ExitThread()来完成这个操作：

```
procedure EndThread(ExitCode: Integer);
begin
  ExitThread(ExitCode);
end;
```

但是线程对象是否析构却取决于 TThread.FFreeOnTerminate 的状态。亦即是说，尽管当前线程已经终结，但 Thread 作为 Delphi 中的一个 TThread 类的对象实例(有可能)仍然未被释放。

这种设计使得程序在运行过程中，可能出现“根本不具有执行能力”的 TThread 对象。例如这样的代码：

```
var
  Thread : TMyThread;
  List : TList;
  i : Integer;

begin
  // ..
  Thread := List[List.add(TMyThread.Create(True))];
  Thread.FreeOnTerminate := False;
  Thread.resume;
```

```
  // ...
end;
```

那在 List 中存在的线程对象将不再有执行能力。因为在访问这些对象时，与它们相关的线程可能已经终结了。

这样设计是有原因的。上文中提到应当将"需要缓存"的线程对象的 FreeOnTerminate 置为 False 值，这样这个 TThread 的子类对象就可以被缓存到一个列表、队列或者栈中。如果这个 TThread 对象仅仅是一个类的对象实例，更确切地说，仅是一个内存块。那么它对操作系统来说，影响当然极小。然而如果并不终结每一个 TThread 对象所对应的线程(操作系统句柄)，并将它们缓存到列表中，那么操作系统可能因为太多的线程被缓存而导致性能下降。

要使得一个被缓存的线程(TThread)对象重新得到执行能力，那么可以将 Create()作为对象方法调用。这同时也会调用到内核例程 BeginThread()，这将导致一个新的线程被分配给该线程对象实例。例如：

```
// 接上例
for i:=0 to List.Count-1 do
  List[i].CreateInual;
```

这两行代码使所有 List 列表中的线程对象重新运行线程执行体。

■ 线程对象的析构

对象的析构总是发生在调用方法 Destory()之后(Free 方法将调用 Destory)：

```
// code from Classes.pas

// 线程对象的析构函数
destructor TThread.Destroy;
begin
  if (FThreadID <> 0) and not FFinished then
  begin
    Terminate;
    if FCreateSuspended then
      Resume;
    WaitFor;
  end;

  if FHandle <> 0 then CloseHandle(FHandle);
  inherited Destroy;
```

```
  FFatalException.Free;
  RemoveThread;
end;
```

如果线程没有中止，析构函数将会试图等待线程正常结束。Delphi 的线程类实现的机制使得这四行代码非常难于理解：

```
// 参见上面的代码
  Terminate;
  if FCreateSuspended then
    Resume;
  WaitFor;
```

Terminate()方法并不实际中止线程，而仅是设置 FTerminated 为值。接下来就调用 WaitFor()方法[1]来等待线程退出。

WaitFor()实际上是等待线程的结束状态，也就是线程调用 ExitThread()发出的信号，而这个 API 实际上是在 Execute()方法执行的末期才被调用的。也就是说，要求线程先处于运行状态。而 FCreateSuspended 表明线程在创建时可能处于挂起状态，因此在 WaitFor()之前会调用 Resume()来唤醒线程。

9.7 线程同步对象

线程同步这个名字令人费解。首先它并不是使被同步的两个线程同时运行——尽管大多数多线程应用看起来是如此。

线程同步确切地说是要解决多个线程之前协调一致的问题[2]。这是一种用于“保证多个线程在访问同一数据时，保持协调一致，避免出错”的机制。

由于两个线程可以是同一进程中的，也可以是跨进程的，因此线程同步机制的设计也涉及到了进程间和进程内的问题。

某一线程中，一行或多行需要不被中断、一次执行完毕的代码称为“关键代码段”。线程同步的目的通常是如下两种：

- 使线程按照一定的次序先后执行，亦即控制多线程的调度次序；

① 该方法在下一节“线程同步对象”中详细讲述。

② Synchronize 的英文原意是“同时发生(To occur at the same time)”和“协调一致(To operate in unison)”两种，中文译作“同步”是很容易令人联想到“同时发生”的。然而多线程中的“同步”这一概念，实则是更偏向于“协调一致”。

- 提供一种使得“关键代码段”能够不被中断的机制。

在思考问题之前，应该记住的是：操作系统在多线程的调度时并不知道哪些代码是“关键代码段”，它只知道哪个线程是在等待的，以及哪个线程可以被激活。

9.7.1　线程同步方法

从操作系统的角度来说，有如下几种线程的同步方法(效率高者居前)，不过应当记住的是，每一种同步方法都是一组 API 的组合使用：

- 原子函数[①]调用。
- 临界区。
- 内核对象等待。

■ 原子函数调用

从汇编一级来讲，一次内存存取并不是对应于一个独立的 CPU 指令。因此即使是使一个整型值加 1 的操作也可能由多条汇编语句组成——当然，也就存在着被中断执行的可能。

原子函数仅用于对内存中的单个值(32 位或 64 位)[②]进行运算，它确保在函数执行过程中不会被中断。原子函数提供的运算包括：

```
// 对整型值递增/递减1，相当于：
//   Addend := Addend + 1;
function InterlockedIncrement(var Addend: Integer): Integer; stdcall;
function InterlockedDecrement(var Addend: Integer): Integer; stdcall;

// 对整型值加一个新值(包括负数)，相当于：
//   Addend := Addend + Value;
function InterlockedExchangeAdd(var Addend: LongInt;
  Value: LongInt): LongInt stdcall;

// 将一个整型值置为新值，相当于：
//   Target := Value;
function InterlockedExchange(var Target: Integer;
```

① 或称为互锁函数。这是同步方法中效率最高的一种方法，因为调用这些 API 时(通常小于 50 个 CPU 周期)，不需要 CPU 从用户方式切换到内核方法(这大约需要 1000 个 CPU 周期)。

② 没有这些函数的 32 位版或 64 位版，是因为操作系统将根据不同的 CPU 来决定如何实现原子函数，而无需用户代码调用不同的版本。

```
 Value: Integer): Integer; stdcall;

// 比较原值如果等于一个值，则用另一个新值替换原值。相当于：
//  if Destination=Comperand then Destination := Exchange;
function InterlockedCompareExchange(var Destination: Pointer;
         Exchange: Pointer; Comperand: Pointer): Pointer stdcall;
```

所有的函数都返回被操作数修改前的原值。

没有原子函数是在不修改原值的情况下读值的 ——因为没有这样的必要。基于原子函数的机制，用户代码直接读取用原子函数修改的值(例如变量)时，总是安全的。

■ 临界区

在用户方式下实现线程同步的第二种方法是使用临界区。执有(或称为“进入”)临界区的线程可以安全地执行代码而不必担心冲突，因为其他欲争用临界区的线程都将被转入(或正在处于)内核模式等待。

```
// 初始化一个临界区对象(记录)
procedure InitializeCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;

// 使当前线程进入临界区
// 如果没有其他线程执有临界区，则继续执行其后代码，否则转入内核模式等待。
procedure EnterCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;

// 当前线程退出临界区
// 一个Enter操作必须有一个对应的Leave操作。
procedure LeaveCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;

// 尝试进入临界区
// 如果有其他线程执有临界区，该API返回false，而并不是等待。
function TryEnterCriticalSection(var lpCriticalSection:
  TRTLCriticalSection): BOOL; stdcall;

// 删除(释放)临界区
procedure DeleteCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;
```

临界区是一个在用户模式下调用的 API，这意味着使用这个 API 仅消耗较少的 CPU。如果尝试进入临界区不成功，则操作系统会使当前线程进入内核模式等待，直到其他线程不再持有

临界区对象之后，才从内核模式切换到用户模式继续执行。在内核模式和用户模式之间切换的操作将导致更多的 CPU 消耗。

理想的情况下，只有较少的线程争用临界区，或者线程只在很短的时间内占用临界区，这样线程就不必“总是”进入内核模式等待。这时系统的效率最高。

较多线程、长时间地占用临界区，会导致系统效率下降，如果某一线程不能从临界区中退出，那么将导致其他线程死锁，再也得不到 CPU 执行权限。

无论如何，临界区总是一种较高效率地实现线程同步的方法，至少比后文中将讨论到的“内核对象等待”高效。

■ 临界区与循环锁

惟一一种既不使用 API，又可以实现同步的方法是使用“循环锁”。这种方法是在 A 线程中循环校验一个变量(例如 LockTag)，而在 B 线程中认为 A 线程应当或能够执行时，将 LockTag 置为 false。这样，A 线程就可以从循环中跳出来执行其后的代码。例如：

```
// A线程
while LockTag do;
// ...

// B线程
LockTag := CheckBox1.Checked;
// ...
```

然而这是绝对应该避免的方法[①]，因为基于 Win32 多线程的机制，A 线程将会大量占用 CPU。而且如果 B 线程优先级较低，则它可能非常难于得到执行代码的权限，这时更多的 CPU 时间就会消耗在 A 线程的那个空循环中——即使此时 CheckBox1.Checked 值已经为 false。

此外，如果有多个 A 线程在空循环中等待，那么当 B 线程将 LockTag 置 false 后，将由操作系统的线程调度来决定接下来哪一个线程跳出循环。这意味着线程同步中的“次序”问题无法解决。

然而操作系统在临界区函数的实现机制上却利用了“循环锁”的机制。这是因为在出现资源争用时，API：EnterCriticalSection()使调用者线程从用户方式切换到内核方式的这个切换过程，所需要的 CPU 时间可能会大于资源被争用的时间。也就是说资源在切换完成之前

① 之所以提到这个失败已极的方法，是因为在处理多线程间数据交叉访问时，这是不熟悉 Win32 机制的开发人员最容易想到的解决方案(这种方案最接近自然思维)。这其实也是我在第一次写多线程程序时犯下的一个错误。

就已经空闲了，那么切换过程(对了，还要切换回来)就白做了。

为此，可以使用下面的 API 来使操作系统在初始化临界区时，指定上述切换之前应当尝试循环测试的次数[①]：

```
// 初始化时设置尝试循环测试次数
function InitializeCriticalSectionAndSpinCount(var lpCriticalSection:
 TRTLCriticalSection; dwSpinCount: DWORD): BOOL; stdcall;

// 调整循环测试次数值
function SetCriticalSectionSpinCount(var lpCriticalSection:
 TRTLCriticalSection; dwSpinCount: DWORD): DWORD; stdcall;
```

很显然，如果在这些尝试中需要占用的 CPU 时间超过两次切换(AB 两线程分别切入切出)所需要的时间总和，那么这个循环次数的设计就是比较失败的。尽管这个 SpinCount 的理想值是可以根据 CPU 速度计算出来的，但对于通常的开发人员来说，只能使用一些经验值(例如 4000)。[②]

上述的这种情况下，如果操作系统发现 A 线程试图进入一个被 B 线程占用着的临界区，那么它大概会做如下的一个运算，以尽可能避免 A 线程进入等待状态：

```
// 1. 从临界区对象中取出SpinCount值(这里假设操作系统用这个域来保存该值)
SpinCount := lpCriticalSection.SpinCount;

// 2. 使用循环锁来检测和尝试进入临界区
repeat
  EnterTag := TryEnterCriticalSection(lpCriticalSection);
  dec(SpinCount);
until EnterTag or (SpinCount<0);

// 3. 如果在指定循环次数之后仍未能进入临界区(仍被占用)，则试图切入内核方式等待
if not EnterTag then
  EnterCriticalSection(lpCriticalSection)

//...
```

■ 内核对象等待

使用内核对象等待的关键是分清楚谁是“等待者”和“被等待者”。例如 A 线程的进一步

① 其实这个循环次数 SpinCount 的设定仅对多 CPU 的系统才有意义，在单 CPU 的环境中，SpinCount 总被忽略而使用默认值 0。至于为什么 Mircosoft 会做这样的设计，是本书惟一留给读者思考的问题(你可以把这看成是软件中的复活节彩蛋)，我仅建议你从线程调度次序的机制这个角度去理解它。

② 这是 Devid Solomon 在《Inside Windows NT》中的推荐值，这可能是因为 AB 两线程分别切入、切出的需要 4*1000 次 CPU 时间。

执行需要 B 线程做某件事，而进程中有一个变量 hEvt 句柄来标识这件事是否完成。那么代码的结构应该是这样的：

```
// 进程中的其他代码(例如主线程)
hEvt := CreateEvent(nil, false, false, 'Test Event');

// 在A线程中：
WaitForSingleObject(hEvt, INFINITE);

// 在B线程中...
// (首先完成一些要做的工作，如果允许A线程继续，则如下调用)
SetEvent(hEvt);
```

这样，A 线程将是“等待线程”，当调用 WaitForSingleObject()后，A 线程停止执行任何代码(失去执行权限)，然后开始等待；B 线程是被等待的，它可以执行任意多的代码，而不用担心 A 线程争用它所操作的资源(因为 A 线程已经停止执行了)。这一直要等到 B 线程调用 SetEvent(hEvt)之后，A 线程才能继续执行。

另外须要注意的是，在 B 线程调用 SetEvent()之后，B 线程并不停止。按操作系统的多线程机制来讲，此后 A、B 两个线程都在同时执行代码，只是两个线程之间不会再有冲突发生[①]。

操作系统所理解的可等待的“内核对象”包括（表 9-1）。

表 9-1　可等待的“内核对象”

内核对象	内核对象的建立(返回句柄)	等待(如下状态发生)	命名[②]	版本
进程	CreateProcess()	进程终止或结束		
线程	CreateThread() CreateRemoteThread()	线程终止或结束		
内存资源通知	CreateMemoryResourceNotification()	可用内存高于或低于临界值[③]		Win2003 WinXP+
文件修改通知	FindFirstChangeNotification()	文件系统发现有修改操作		

① 是否有冲突是编程过程中开发人员的预估，事实上的运行状态也许并非如此，因为操作系统并没有(自动)处理线程之间的冲突的机制。一旦有冲突发生，而开发人员又没有预估到这种状况并实现处理代码，那么线程和进程就将崩溃。预估到所有可能的资源争用，并通过“同步机制”来避免冲突的发生是多线程应用开发中需要仔细分析的问题。

② 内核对象在多个进程间使用同一个命名空间，因此如果试图创建或者打开一个在其它进程中命名过的对象，且当前进程没有目标进程的权限，就会导致出错。

③ 临界值由操作系统设定。这个等待对象有助于编写一个与内存分配有关的伺服线程：当可用内存不够(low)时，将大量多余内存块释放或转储到硬盘；充足(high)时，分配更多内存。

续表

内核对象	内核对象的建立(返回句柄)	等待(如下状态发生)	命名	版本
控制台输入	CreatFile()[1] GetStdHandle()	有控制台输入操作		
设备对象	文件，套接字，通讯端口的句柄	被等待对象完成异步I/O (I/O请求处理完毕)		
作业	CreateJobObject()	作业的时间结束	能	Win2000
可等待定时器	CreateWaitableTimer()	指定时间到达	能	WinNT+
互斥(信号灯)	CreateMutex()	被等待线程调用API: ReleaseMutex()	能	WinNT+
信标(信号量)	CreateSempahore()	内置计数值大于0[2]	能	
事件	CreateEvent()	被等待线程调用 API: SetEvent()	能	

内核对象等待 API 则总是在内核模式下执行的，这与使用临界区确实存在一些不同，尽管二者都可能导致线程等待。使用内核对象等待的另一个理由是跨进程的线程同步。例如 A 线程要对另一个进程中的 B 线程(可以是主线程)的文件修改操作做出响应。类似的功能是常见的：在 Delphi IDE 之外用记事本编辑文件并存盘，那么 IDE 会弹出这个对话框（图 9.7）。



图 9.7 IDE 弹出对话框

这样的功能的实现代码其实非常简单[3]，只须在 A 线程中创建一个线程并调用等待函数：

① 仅当使用 CreateFile()并指定'CONIN$'为文件名时，将返回一个控制台输入句柄。

② 信号量是惟一一个允许多个线程同时占用资源的同步函数。CreateSemaphore()可以设置内置计数器的初值与最大值，调用等待函数时将导致计数器值减一，而 ReleaseSemaphore()将使计数器值加一。计数器中的值表明"还允许并行运行的线程数"，显然该值为 0 时表明当前线程需要进入等待状态。

③ 下面的例子中用 FindFirstChangeNotification()来实现目录监视并不是最佳方案(不能取得被变更的文件详情)，较好的方案是使用文件异步 I/O 作为同步对象。完整代码参见随书光盘。

```
var
  DirName : string = '';

function NotificationProc(Param:Integer);
var
  hHandle : THandle;
begin
  hHandle := FindFirstChangeNotification(PChar(DirName), true,
    FILE_NOTIFY_CHANGE_FILE_NAME);
  try
    WaitForSingleObject(hHandle, INFINITE);
    ShowMessage('hi!');
    // and more...
  finally
    CloseHandle(hHandle);
  end;
end;
```

大多数情况下，伺服线程须等待多个对象之一发出信号。这时可以使用 API：WaitForMultipleObjects()。假如上例须等待多个目录的修改事件(输出信息)，则可以这样设计：

```
// ...

var
  CreateNotifyEvent : THandle;

procedure TWatchThread.Execute;
var
  Id : Integer;
  WAIT_OBJECT_N : Integer;
begin
  // ...
  // hHandles[0] := CreateNotifyEvent;
  repeat
    // 等待，当任意对象(句柄)发生状态信号变化时返回
    WAIT_OBJECT_N := WaitForMultipleObjects(Length(hHandles),
      @hHandles[0], false, INFINITE);

    // 出错?
    if WAIT_OBJECT_N = WAIT_FAILED then break;

    // 计算对象句柄在数组中的索引
    WAIT_OBJECT_N := WAIT_OBJECT_N - WAIT_OBJECT_0;
```

```
if WAIT_OBJECT_N = 0 then
begin
  ResetEvent(hHandles[0]);

  // 添加一个新的监视记录
  // Id := Length(hHandles);
  // SetLength(hHandles, Id+1);
  hHandles[Id] := FindFirstChangeNotification(
    PChar(WatchTarget[Id]),
    true,
    FILE_NOTIFY_CHANGE_FILE_NAME or
    FILE_NOTIFY_CHANGE_DIR_NAME or
    FILE_NOTIFY_CHANGE_LAST_WRITE);

  if hHandles[Id] <> INVALID_HANDLE_VALUE then
    // 添加成功
  else
    // 添加失败
end
else
begin  // 错误否?
  if WAIT_OBJECT_N >= Length(hHandles) then Continue;

    // 目录有在修改, 显示信息
    // Synchronize(ShowNotifyInfo);

    // 继续监视 hHandles[WAIT_OBJECT_N] 句柄所指的对象
    FindNextChangeNotification(hHandles[WAIT_OBJECT_N]);
  end;
until Terminated;
end;
```

当主线程中成功地选择一个待监视目录后，可以置 CreateNotifyEvent 对象状态，这时 WatchThread 的等待结束，然后进入“WAIT_OBJECT_N = 0”的分支中添加一个新目录并监视。

9.7.2 Delphi 中的线程同步类

在 *SyncObj.pas* 单元中，Delphi 实现了临界区和等待对象同步两种线程同步类。它们都继承自基类 TSynchroObject，如图 9.8 所示。

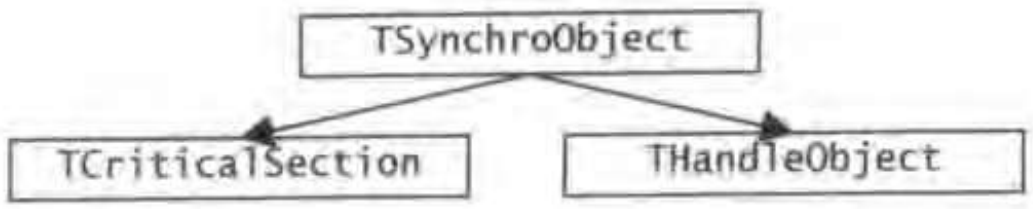


图 9.8　线程同步类与 TSynchroObject 的继承关系

其中，临界区 TCriticalSection 类是对 API 的一个简单封装，而 THandleObject 则是对“可等待内核对象的句柄(Handle)”的进一步抽象。虽然 THandleObject 不是抽象类，但也不能够创建该类的对象实例来使用，而应当使用它的派生类（图 9.9）：

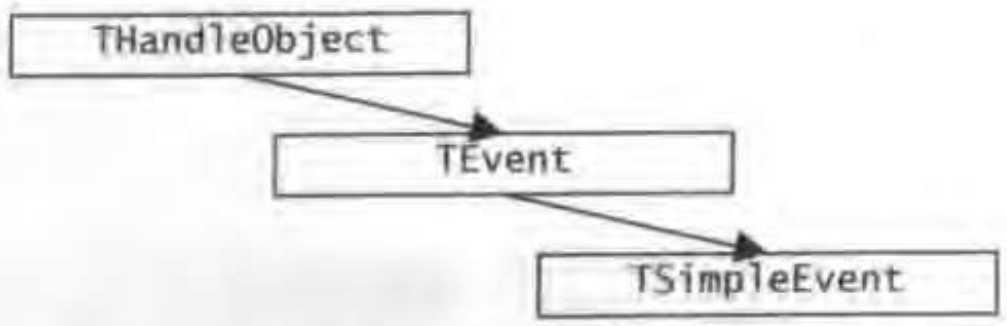


图 9.9　THandleObject 的派生类

TEvent 是基于操作系统的同步概念，在 *SyncObj.pas* 单元中实现的“事件同步对象(Event)”类，它的构造函数直接封装了 API：CreateEvent()：

```
// code from SyncObjs.pas
constructor TEvent.Create(EventAttributes: PSecurityAttributes;
  ManualReset, InitialState: Boolean; const Name: string);
begin
  FHandle := CreateEvent(EventAttributes, ManualReset, InitialState,
    PChar(Name));
end;
```

TSimpleEvent 则重写 TEvent 的 Create() 方法，使得事件对象被创建后即处于无信号、手动复位(ManualReset)状态：

```
// code from SyncObjs.pas
constructor TSimpleEvent.Create;
begin
  inherited Create(nil, True, False, '');
end;
```

可以认为事件对象与临界区对象是类同的：它们都是使当前线程暂停执行并等待另一线程做出响应。二者的区别主要在于：临界区用于当前进程中的多线程同步，而 Event 对象则可以

实现跨进程的线程同步，这是仅在 *SyncObj.pas* 单元中实现 TEvent 和 TCriticalSection 类的原因：只需要这两个对象就可以解决进程内和进程间的线程同步问题[①]。

在实际应用中，仅解决线程本身的同步是不够的："线程"的粒度太大，以"线程"作为同步的"基本单位"导致效率过低。因此，在 *SysUtils.pas* 单元中，Delphi 使用临界区(CriticalSection)和事件(Event)来分别实现下列存取同步类：

- TSimpleRWSync。
- TMultiReadExclusiveWriteSynchronizer(TMREWSync)。

存取同步类仅解决对"同步存取某一资源"的操作进行同步，以"操作"作为同步单位当然比"线程"来得小，因而效率也可以更高[②]。

以"操作"作为同步单位的另一个理由，是"操作"(作为一种行为)可以有附加的逻辑，而"内核对象"本身就是实现的结果(或目标)，它没有附加逻辑的可能。因此，上面的"RWSync(Read/Write)同步"可以附加一个"多读单写"的逻辑，从而衍生出"MREWSync(多读单写)同步"这样的类设计（图 9.10）：

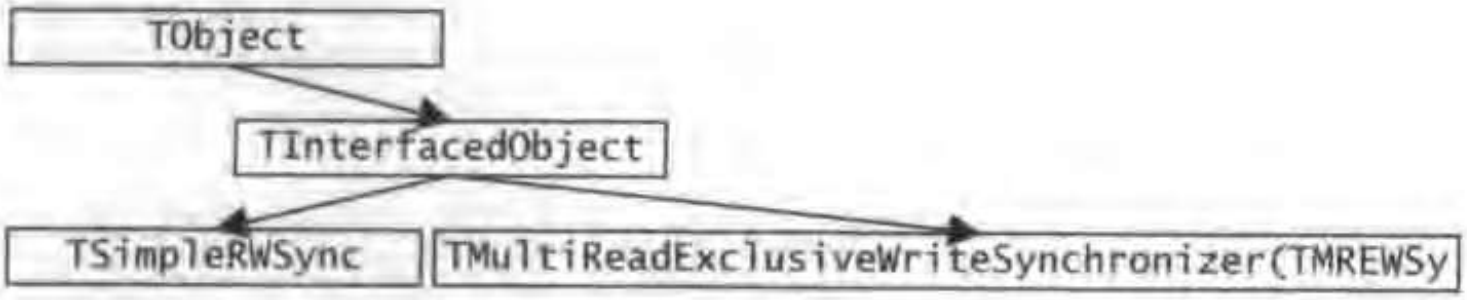


图 9.10

在 Delphi 内置支持的同步方法仅有上述几种，这可以满足绝大多数情况下的线程同步需求。在应用系统中，可以参考 TEvent 的代码，从 THandleObject 派生同步其他内核对象的类；或者参考 TMREWSync 来实现用户系统中的同步逻辑。

TMREWSync 的实现相对要复杂得多，由于它允许同一线程多次调用 BeginWrite()方法(只要该线程也调用相同次数的 EndWrite()方法)，因此需要一个"写递归计数器"来记录这

① 通常的资料中会将 Mutex 和 CriticalSection 作比较，它们是大致相同的。因此 Delphi 应当也可以在 *SyncObj.pas* 中实现 TMutex 而不是 TEvent，但 Mutex 功能相对较弱，反复的 CreateMutex()和 ReleaseMutex()效率也不高，所以 *SyncObj.pas* 中选择实现 TEvent。

② 这也是操作系统实现其它"可等待内核对象"的原因之一。将"操作"对象(例如文件或者 I/O 状态)作为"同步"的对象，不会如同同步"线程状态"这样大粒度的对象一样使效率变低。

个值，直到该值为0，则表明另一个线程可以开始读操作。这个值应当在TMREWSync的字段中，与具体线程无关：有且仅有一个写线程可以执有该计数器，该线程ID被记录在TMREWSync.FWriterID域中：

```
// code from SyncObjs.pas
 TMultiReadExclusiveWriteSynchronizer = class(TInterfacedObject,
   IReadWriteSync)
 private
   // ...
   FWriteRecursionCount: Cardinal;          // 写递归计数器
   tls: TThreadLocalCounter;                // 执有该TMREWSync的线程列表(读&写线程)
   FWriterID: Cardinal;                     // 写线程ID
```

每个线程应该有一个自己的RecursionCount域，用于记录它作为读线程时的BeginRead()请求次数。因为它是与每个线程相关的，所以应该被声明为threadvar，以使得它被保存在TLS中。但是限于TLSArray的大小，在Windows 2000以下的操作系统中，进程可以同时使用TLS的最大模块数量为64个。因此TMREWSync类中使用一个TThreadLocalCounter类型的tls字段替代了操作系统的TLS来避免这样的问题。

TThreadLocalCounter(线程本地计数器)类并不是要统计当前进程中的TThread对象实例的个数，而是记录所有执有该TMREWSync的线程。每个线程可以通过tls.Open()来取得一个对应于当前线程的ThreadInfo(线程记录块)，而读线程递归计数器被放在TThreadInfo的RecursionCount域：

```
// defined in SyncObjs.pas
 TThreadInfo = record
   Next: PThreadInfo;
   ThreadID: Cardinal;                      // 对应的线程ID
   Active: Integer;                         // 如果值等于Alive，表明结点在用

   // 读递归计数。如果RecursionCount>1，表明当前线程递归读取。
   RecursionCount: Cardinal;
 end;
```

如果当前进程的多个线程需要使用REWS机制同步某个资源，则进程中应当有一个TMREWSync对象。每个线程都通过TMREWSync的BeginRead()来得到该资源的读权限，通过BeginWrite()来得到写权限，当然也应该通过EndRead()和EndWrite()来结束读写操作。TMREWSync中使用两个事件对象来标识对资源的读写状态：

```
// code from SyncObjs.pas, TMultiReadExclusiveWriteSynchronizer.Create()
// ...

  // manual reset, start signaled
```

Delphi源代码分析

```
FReadSignal := CreateEvent(nil, True, True, nil);
// auto reset, start blocked
FWriteSignal := CreateEvent(nil, False, False, nil);
```

如果资源可读，那么(在无论任意多个线程中的)单一线程中多层递归读取该资源，都是允许的。在这样的存取逻辑中，线程在第一次请求读权限时可能因为有写线程而挂起。然而一旦取得读权限(可能是从挂起中返回或者直接得到读权限)，线程就只需要操作读递归计数器(TThreadInfo.RecursionCount)了，并不须要再进入等待状态(因为在读过程中不会有写线程在运行)。因此 BeginRead() 方法这样实现：

```
// code from SyncObjs.pas
procedure TMultiReadExclusiveWriteSynchronizer.BeginRead;
var
  Thread: PThreadInfo;
  WasRecursive: Boolean;
  SentValue: Integer;
begin
  tls.Open(Thread);                                    // 取线程信息块
  Inc(Thread.RecursionCount);                          // 读计数
  WasRecursive := Thread.RecursionCount > 1;           // 是否在同一线程中多次读数

  if FWriterID <> GetCurrentThreadID then              // 是否是写线程
  begin
    if not WasRecursive then                           // 如果不是写线程，且是第一次读数(非递归)
    begin
      WaitForReadSignal;                               // 等待读信号
// ...
```

如果当前线程是写线程，则可以任意读取资源；如果不是写线程，且是第一次使用 BeginRead() 来访问资源，那么当前线程须等待读信号。如果其他线程在写资源，则挂起线程等待，否则 API：WaitForReadSignal() 不会挂起当前线程，而进入其后的代码：

```
// 接上面的代码
// ...

    // 当FSentinel原始值<=0
    while (InterlockedDecrement(FSentinel) <= 0) do
    begin
      SentValue := InterlockedIncrement(FSentinel);  // 恢复FSentinel值
      if SentValue = mrWriteRequest then             // 如果 FSentinel - 1 = $FFFF
        UnblockOneWriter;                            // 置允许写信号
      Sleep(0);                                      // 让出执行权限，唤醒在等待的写线程
      WaitForReadSignal;                             // 等待读信号
    end;
```

这里要解决的是大量的读线程可能导致写线程总是处于等待状态的问题，亦即“写线程(写者)饥饿”[①]。“让出执行权限并唤醒在等待的写线程”的操作，只可能发生在读操作的递归之外，也就是第一次发生读请求的时候。这是避免在递归过程中读取到一个“变化的”资源，按理来说，递归过程中读同一资源的值应当是惟一的。

这只是在表面上看起来合理。做一个假设，如果A线程处于递归读状态，B线程请求写并且已经饥饿，C在申请读时，就会因为发现写线程B饥饿而让出权限，并且C线程挂起等待。这时，A线程由于处于递归状态，因此不需要检测FReadSignal就可以读，而B线程由于已经得到执行权限也可以写。那么，接下来的问题是：系统中出现了读、写线程可同时操作同一个资源的情况。

TMultiReadExclusiveWriteSynchronizer(TMREWSync)类在Delphi7中的实现仍然是有BUG的，线程饥饿的处理并不十分合理。但这并不影响TThreadLocalCounter的对象的实现。尽管以目前的实现情况来看，这个类是专门留给TMREWSync使用的，但开发人员也可以在自己的代码中用它来替代TLS：

- 将被保护数据放到ThreadInfo块(记录)的某个域中；
- 存取当前线程中受保护的数据结构之前，应当先调用Open()方法打开当前线程的ThreadInfo块；
- 每个Open()调用在当前线程终结之前，应当有一个匹配的Close()操作(或Delete()操作)；

可以使用派生TThread的子类并设计与该子类相关的字段来实现与TLS类似的机制[②]。因此使用TThreadLocalCounter的惟一理由是：它与某个具体的TThread对象无关，它识别线程时使用的是操作系统线程对象的ThreadID。

9.7.3　线程对象的WaitFor()方法

线程类TThread的WaitFor()方法被用于停止当前线程并等待TThread所对应的线程对象。例如AThread是一个新线程，那么在B线程中调用AThread.WaitFor()就会使B线程停止运行并等待，直到AThread结束或者中止。

然而这样的设计存在例外的情况。如果主线程调用AThread.WaitFor()，那么主线程一旦挂起，将导致主线程无法再响应任何消息，其他线程(包括AThread)的Synchronize()方法也将无法被执行。这样就出现了死锁：主线程在等待AThread结束，而AThread和其他线程

① 线程长时间处于等待状态的问题被称为“饥饿”。

② 使用与TThread实例相关的字段而非操作系统的TLS机制，是官方推荐的方法。

在等待主线程调用被同步的方法。

因此在TThread.WaitFor()中使用了一点小的技巧来避免这样的情况发生：

```
// code from Classes.pas, function TThread.WaitFor()
// ...

 H[0] := FHandle;
 if GetCurrentThreadID = MainThreadID then
 begin
   WaitResult := 0;
   H[1] := SyncEvent;

   // 如果当前线程是主线程，则循环直到 Self.FHandle 所指向的线程结束或中止。
   repeat
     // 如果等待中发现第三个被等待对象(QS_SENDMESSAGE标志)状态改变，则处理消息
     if WaitResult = WAIT_OBJECT_0 + 2 then
       PeekMessage(Msg, 0, 0, 0, PM_NOREMOVE);

     // 开始等待三个输入对象(第三个对象是dwWakeMask值，本例中是
     // QS_SENDMESSAGE)。如果等待超过1秒(入口参数中的超时值为1000ms)，
     // 则从等待状态中切出并尝试下一次等待。
     WaitResult := MsgWaitForMultipleObjects(2, H, False,
       1000, QS_SENDMESSAGE);
     CheckThreadError(WaitResult <> WAIT_FAILED);

     // 如果等待中发现 SyncEvent 状态改变，则在当前线程(主线程)中调用
     // CheckSynchronize 来执行同步方法，然后继续等待 Self.FHandle
     // 所指向的线程的终结。
     if WaitResult = WAIT_OBJECT_0 + 1 then
       CheckSynchronize;
   until WaitResult = WAIT_OBJECT_0;
 end
 else
   // 如果不是主线程，则直接等待 Self.FHandle 所指向线程的终结。
   WaitForSingleObject(H[0], INFINITE);
// ...
```

在ThreadProc()的结束代码以及类方法TThread.Synchronize()中总是会调用例程SignalSyncEvent()：

```
// code from Classes.pas
procedure SignalSyncEvent;
begin
  SetEvent(SyncEvent);
```

```
end;
```

因此当线程类 TThread 的实例调用(或直接类方法调用) Synchronize() 方法，或 ThreadProc()执行结束都将使处于等待中(WaitFor)的主线程得到一次处理的机会。

9.8　多线程环境下的内存管理器

无论如何，多线程下的内存管理器的效率将只有正常效率的 1/8[1]，这已然成为事实。现在只是大多数开发人员并不知道它的原因而已。

在 *GetMem.inc* 中的下面几行代码可探究到这种现象的根源：

```
// code from GetMem.inc, function BlkGetMem()
// ...

  try
    if IsMultiThread then EnterCriticalSection(heapLock);
    // ...
  finally
    if IsMultiThread then LeaveCriticalSection(heapLock);
  end;
end;
```

在多线程中，作为操作系统资源的内存总是在受保护的状态下进行分配。这导致在大量的线程中频繁分配内存时，多个线程间常常处于等待状态，大量的 CPU 被消耗在进入和退出内核模式的系统切换操作中。

在多线程各个线程中都进行内存分配的确是一个不好的主意。但是 Delphi 的 *GetMem.inc* 使得用户代码根本没有办法控制局面：如果开发者使用 *System.pas* 中的系统例程或 TThread 类创建了新线程，那么系统就立即进入 IsMultiThread 的模式，而无论开发者是否真的要在多个线程中分配内存。

垃圾收集器(GC线程)是解决这个问题的思路之一。如果使用独立的线程来分配和释放内存，在必要的时候才进行一次垃圾清理，效率可以高很多。但如果并发太多的线程，或者系统总是在忙，垃圾将迟迟不能回收(或者回收过程太长)，那么效率也是较低的。

可以使用第三方的内存管理器来提高效率，但这只是总体效率的提升。例如 QMemory 可以

① 这还取决于线程的数量以及分配内存块的大小、频度，在一些测试中，这个值是 1/6-1/12。

使单线程情况下(比缺省内存管理器)效率提升 30%~50%左右，在多线程模式下效率也可提高相应的比例。但是多线程时仍比单线程时慢很多。原因是第三方内存管理器仍然使用与 *GetMem.inc* 同样的方式(临界区)来实现多个线程中内存管理器的同步。

共享内存管理器(*ShareMem.pas* 和 *borlandmm.dll*)缺省总是将 IsMultiThread 置为 True，这使得在使用共享内存管理器的应用中，无论是否真的多线程，系统的内存管理效率总是处在非常低效的状态中。使用 FastShareMem 与 ShareMemRep 这两个第三方的共享内存管理器时，IsMultiThread 却是处于 False 状态的，这样可以避免上述的问题。不同的是，ShareMemRep 有自动识别当前系统是否处于多线程(并重置 IsMultiThread 值)的机制，而 FastShareMem 没有。

在多线程方式下使内存管理高效的另一种方法是避免使用 Delphi 的内存管理器。比较常规的技术可能包括：

- **减少对象、动态数组、长字符串和宽字符串在线程中的使用**
 例如可以使用 TLS 或者 TThread 对象相关字段在线程中缓存上述数据结构，从而避免每次进入例程都需要分配内存。或者，也可以使用缓存池来建立自己的缓存机制。
- **使用栈分配来替代堆分配**
 例如使用记录来代替数组、使用静态数组来代替动态数组，或者使用短字符串来代替长字符串。这些数据类型在进入例程时都在栈上分配，退出时也通过栈修正来自动释放，效率远高于被替代的数据类型。
- **从 Delphi 的内存管理器中一次性开辟大内存块并自行管理**
 如果需要频繁分配和释放内存，可以先开辟一大块内存然后自行管理，以保证 Delphi 的内存管理器总是运行在单线程模式下。这个大内存块可以用作字符串、字符串数组或者频繁使用的 Buffer。

如果能确保只有一个线程在使用内存管理器，那么的确可以在多线程应用的代码中手工将 IsMultiThread 值设为 False。由于 Delphi 的源代码中，仅有少数几处使用到了 IsMultiThread 这个变量：

- *$(Source)\RTL\Corba45\Corba.pas*：总是置 IsMultiThread 为 True，以确保内存管理器是线程安全的；
- *$(Source)\RTL\Common\ComObj.pas*：需要检测 IsMultiThread 值以确认当前系统是否处在多线程状态；
- *$(Source)\VCL\Ib.pas*：总是置 IsMultiThread 为 True；
- *$(Source)\WebSnap\WebScriptAS.pas*：需要检测 IsMultiThread，也总是会使用 *ComObj.pas* 单元。

因此在大多数情况下，这不会带来麻烦。开发者只要确定没有在项目(包括项目中使用的第三方控件或者代码)中引用上述单元，那么就可以使用自己的 IsMultiThread 标志来替代这个系统缺省值：

```
// 用户工程文件
program UserProject;
uses
  SystemAdjuster,
// ...
end.


// 重置 IsMultiThread 标志变量
// 可以在代码中用 _IsMultiThread 变量随时访问真正的 System.IsMultiThread
unit SystemAdjuster;
interface
var
  IsMultiThread : Boolean;
  _IsMultiThread : Boolean absolute System.IsMultiThread;

implementation

initialization
  // 保存初始时系统的多线程状态(DLL中有机会在主模块的单元初始化前调整这个值)
  IsMultiThread := System.IsMultiThread;
  // 让系统(内存管理器)总是运行在单线程模式下
  System.IsMultiThread := False;

end.
```

第 10 章

资源的管理和使用

Delphi 使用编译指令{$R}来包含资源到编译后的文件中，同时它支持用 resourcestring 关键字在代码中声明存放在资源节中的字符串。对于前者来说，编译器仅是将资源放到文件的资源节(.rsrc)中而已。而对于后者，Delphi 加入了大量的代码，使开发人员可以用类似于字符串常量的方法来操作字符串资源。

除了资源字符串之外，绝大多数资源应当使用专用的 API 来读取。对于 Delphi 的内核来说，也是使用标准 Win32 API 来操作字符串资源的。

10.1 资源字符串

对于一个使用资源字符串的项目，Delphi 一般按照如下的规则来编译代码：

- 如果使用了 resourcestring 关键字定义资源字符串，则总是按照 PE 文件结构中有关字符串表的存储规则，将字符串编译到资源节中。
- 如果在代码中使用到了资源字符串，则在代码前加入一个 LoadResString()调用。
- 如果在常量、变量定义中使用到了资源字符串，则在当前单元的初始化代码的最前面加入一个_InitResStrings()调用。

10.1.1 隐含的调用：LoadResString()例程

每次引用一个资源字符串，Delphi 都会先隐含地声明一个 AnsiString 类型的变量，然后调用 LoadResString()将资源字符串转储到字符串变量中。如下例：

```
resourcestring
  S_TestRes = 'This is a test string!';

var
  str : string;
begin
  str := 'Hi, ' + S_TestRes;
end.
```

在反汇编代码中可以看到对例程 LoadResString()的调用：

```
Project1.dpr.14: str := 'Hi, ' + S_TestRes;
00407D6F 8D55EC          lea edx,[ebp-$14]   // 在栈上的一个临时字符串变量

// 指向 S_TestRes 的PResStringRec 记录
00407D72 B8047C4000      mov eax,$00407c04
00407D77 E864C4FFFF      call LoadResString // LoadResString(eax, edx)①
00407D7C 8B4DEC          mov ecx,[ebp-$14]  // 返回临时字符串在[ebp-$14]
00407D7F B898974000      mov eax,$00409798
00407D84 BAB87D4000      mov edx,$00407db8
00407D89 E8267BF9FF      call @LStrCat3     // str := str + 临时字符串
// ...
```

从这里可以看出，资源字符串的实现与有初值的字符串变量和字符串常量在代码节（CODE）中的实现有类同之处。只不过，在 CODE 节中的字符串是通过引用计数为-1 来做标识，以使得每次引用之前产生一个字符串的拷贝；而在资源节（*.rsrc*）中字符串却是通过在代码中插入 LoadResString()例程代码，从而在使用前产生一个新的字符串变量。

因此直接在代码中动态地使用资源字符串是效率偏低的做法：因为每次使用时都会声明隐含的变量并调用 LoadResString()。因此下面两段代码的效率并不一致（访问资源字符串时并不使用引用计数机制）：

```
resourcestring
  ResString = 'Test Resource String!';

var
  i : integer;
  str : string;
  list : TStringList;
```

① 这里看起来与 LoadResString()的声明并不相同，这是因为编译器应用了另外的一条规则：使用追加参数的方式传出字符串返回值。关于这一规则，可参见第 3 章第 7 节的“汇编例程中的返回值约定”。

```
// ...

// 低效的代码段一
  for i := 0 to 1000 do
    list.Add(ResString);

// 高效的代码段二
  str := ResString;
  for i := 0 to 1000 do
    list.Add(str);
```

在 *System.pas* 中，LoadResString()例程的实现代码并不复杂。但有一点是需要记住的，入口参数的 ResStringRec 通常是由 Delphi 在编译期生成并硬编码在代码中的：

```
function LoadResString(ResStringRec: PResStringRec): string;
var
  Buffer: array [0..1023] of char;
begin
  if ResStringRec = nil then Exit;
  if ResStringRec.Identifier < 64*1024 then
    SetString(Result, Buffer,
      LoadString(FindResourceHInstance(ResStringRec.Module^),
        ResStringRec.Identifier, Buffer, SizeOf(Buffer)))
  else
    Result := PChar(ResStringRec.Identifier);
end;
```

在上述代码中，API：LoadString()根据 ID（Identifier）从指定实例的资源节中调入字符串，并返回从资源节复制到 Buffer 中的字符串的长度[①]。SetString()则将 Buffer 中的字符串复制到 Result 中。

10.1.2 _InitResStrings()例程

如果资源字符串不是在运行期动态请求的，而是在源码中就声明引用的（例如声明一个常量或有初值变量来提高使用资源字符串的效率），那么编译器就会在单元初始化中隐含一个变量初始化过程。例如：

① 受限于 Buffer 数组的大小，Delphi 不能自动处理长度大于 1024 字节的资源字符串。如果使用这样的字符串，需要另行使用 API 将资源调入字符串变量中。这个 BUG 在 Delphi 7.1 Update 中被得到了修正。

```
resourcestring
  S_ErrorOfUserInterface = 'Test Resource String!';

// 声明一个资源字符串的引用，声明有初值的变量时与此相同。
const
  ErrStr_UserInterface : string = S_ErrorOfUserInterface;
```

类型化常量 ErrStr_UserInterface 在编译期声明，并在数据节（DATA）中占有 SizeOf(AnsiString)字节，但它的初始值为0。直到单元初始化时，才会在堆中分配内存，并将数据复制到堆中。完成这件工作的即是_InitResStrings()例程。

在_InitResStrings()入口处，编译器传入需要初始化的资源字符串的列表（StrInitTable），该列表结构为：

```
// code referenced by System.pas, _InitResStrings()

TResStringInitItem = record
  variableAddress: String;
  resStringAddress: PResStringRec;
end;

TResStringInitTable = record
  cnt : Integer;
  tab : array [0..0] of TResStringInitItem;   // 0 .. cnt - 1
end;
```

_InitResStrings()列举表中所有初始化项目，然后调用 LoadResString()。

```
procedure _InitResStrings;
// ...
@@loop:
        MOV     EAX,[ESI+4]     { load resStringAddress     }
        MOV     EDX,[ESI]       { load variableAddress      }
        ADD     EAX,EBX
        ADD     EDX,EBX
        CALL    LoadResString
// ...
```

上面的循环代码每次完成的操作相当于：

```
// next call from _InitResStrings()
StrInitTable.tab[i].variableAddress := LoadResString(
  StrInitTable.tab[i].resStringAddress);
```

因此在完成_InitResStrings()调用后，@StrInitTable.tab[i]就指向了一个有效的字符串。对于编译器来说，只需要在代码中编码@StrInitTable.tab[i]这个地址即可，而不再需要每次访问资源字符串时都调用 LoadResString()。

除此之外，与资源字符串有关的还有如下两个内部例程：

```
procedure _InitResStringImports;
procedure _InitImports;
```

10.1.3 _InitResStringImports()例程

例程_InitResStringImports()与_InitResString()的作用是类似的，只是当资源字符串被定义在其他单元中时，才会调用_InitResStringImports()例程来初始化。例如：

```
uses
  SysConst;  // <- 资源字符串 SLongDayNameSun定义在 SysConst 单元中

resourcestring
  S_ErrorOfUserInterface = 'Test Resource String!';

const
  // 调用 _InitResStrings() 初始化
  LocalStr_UserInterface : string = S_ErrorOfUserInterface;

  // 调用 _InitResStringImports() 初始化
  ImportStr_LongDayNameSun : String = SLongDayNameSun;
```

Delphi 使用不同的例程来处理位于单元内、外的资源字符串的初始化，主要是考虑运行期包的使用。这很容易理解：在编译期，Delphi 并不能确知运行期包的载入地址。因此编译器为_InitResStringImports()准备的初始化表与_InitResStrings()略有不同：

```
// code referenced by System.pas, _InitResStringImports()

PResStringInitItem = record
  variableAddress: String;
  resStringAddress: ^PResStringRec;
end;

PResStringInitTable = record
// ...
// 参考 _InitResStrings() 例程
```

resStringAddress 域被定义成了一个指向 PResStringRec 的指针。对于当前单元中的字符串_InitResString()，这个指针可以直接指向一个资源字符串的记录，因为这些资源数据肯定在当前模块的资源节（*.rsrc*）中；而其他单元中的字符串可能位于一个外部的模块（例如运行期包）中。

```
procedure _InitResStringImports;
// ...
@@loop:
        MOV     EAX,[ESI+4]     { load address of import        }
        MOV     EDX,[ESI]       { load address of variable      }
        MOV     EAX,[EBX+EAX]   { 在后文中将进一步描述...        }
        ADD     EDX,EBX
        CALL    LoadResString
// ...
```

由于 EBX+EAX 表明 item.resStringAddress，因此 [EBX+EAX] 相当于 item.resStringAddress^，亦即是指向外部的模块中该资源字符串的 PResStringRec 记录。这个记录地址被直接传入 LoadResString()，因此记录在外部模块中如何重定位是没有考虑的。

这是一个精巧的、别出心裁的设计。在_InitResString()中，因为访问资源节中的地址时需要被重定位，所以编译器为例程传入的 TResStringInitTable.tab 中的元素（Item 的地址）也是需要被重定位的。然而（在使用运行期包时，）编译器为_InitResStringImports()所传入的 TResStringInitTable.tab 中的元素就不需要重定位，只需要将它指向 PE 文件的导入表（*.idata*）中的某个项，就可以让操作系统在载入 PE 文件时完成相应地址的修正。

除了处理 TResStringInitTable.tab 时的细小差异，_InitResStringImports()的实现代码与_InitResStrings()是几乎一致的。

10.1.4　_InitImports()例程

如果引用在其他单元中的资源字符串的地址，则会调用_InitImports()初始化过程。下面的代码展示相关代码的一些区别：

```
uses
  SysConst;  // <- 资源字符串 SLongDayNameSun定义在 SysConst 单元中

resourcestring
  ResStr = 'Test Resource String';
```

```
var
  p1 : Pointer = @ResStr;                  // 不需要初始化过程，将直接存取资源节.rsrc
  p2 : Pointer = @SLongDayNameSun;         // 调用 _InitImports() 初始化
  V_Str : String = ResStr;                 // 调用 _InitResString() 初始化
  V_ImportStr : String
              = SLongDayNameSun;           // 调用 _InitResStringImports()初始化
```

在这个例子中，由于指针 p2 初值指向在其他单元中的资源字符串的地址，因此在初始化时会为变量 p2 调用_InitImports()。

然而这只是一个表面现象：事实上这是与资源字符串无关的。无论代码中 p2 访问的是@SLongDayNameSun，还是在其他单元中定义的其它数据的地址，都将导致调用_InitImports()操作的发生。

_InitImports()存在的理由与_InitResStringImports()是一致的：导入运行期包的导出例程或者符号的地址。略有区别的是，_InitResStringImports()针对资源字符串本身，而_InitImports()针对一个目标模块导出的地址（当然也包括资源字符串地址）。因此两个函数的实现代码差异不大。但编译器为二者传入的初始化表并不一致：

```
// code referenced by System.pas, _InitResStringImports() and _InitImports()

// for _InitResStringImports()
TResStringInitItem = record
  variableAddress: String;
  resStringAddress: ^PResStringRec;
end;

// for _InitImports()
TImportItem = record
  variableAddress: Pointer;
  sourceAddress: ^Pointer;
  sourceOffset: LongInt;
end;

// 以下参考 _InitResStrings() 例程
TResStringInitTable = record
// ...

TImportInitTable = record
// ...
```

之所以 TResStringInitItem 与 TImportItem 有这样细微的差别，是因为资源字符串是通过 API 来访问 PResStringRec 结构中的 Identifier 以载入数据的，因此并不关心数

据在目标模块中的具体位置；而导入地址时需计算目标地址，因此加入了一个移偏值 sourceOffset。这个区别导致两个例程的实现代码的惟一处不同：

```
procedure _InitResStringImports;
@@loop:
        // ...
        ADD    EDX,EBX          // EDX := EDX + EBX, 计算运行期的variableAddress
        CALL   LoadResString    // 载入资源串到新创建的字符串变量中，返回变量地址到EDX
        ADD    ESI,8  // ESI := ESI + SizeOf(TResStringInitItem), 下一个item
        DEC    EDI    // TResStringInitTable.cnt - 1, 列举tab数组时的计数
        JNZ    @@loop
// ...

procedure _InitImports;
// ...
@@loop:
        // ...
        // -->   EAX: item.sourceAddress^
        //       ESI: @item

        // 计算目标模块中的变量地址：EAX :=sourceAddress^ + Offset
        ADD    EAX,[ESI+8]

        // 将EAX值送入到variableAddress，隐含了 variableAddress 运行期地址的计
        // 算(参考上一段代码中的EDX := EDX + EBX)
        MOV    [EBX+EDX],EAX
        ADD    ESI,12           // ESI := ESI + SizeOf(TImportItem), 下一个item
        DEC    EDI              // TImportInitTable.cnt - 1, 列举tab数组时的计数
        JNZ    @@loop
// ...
```

10.2　Delphi 编译的 PE 文件的默认资源

对于 Delphi 编译生成的模块，编译器会自动的处理其 PE 头结构中的资源节。并以固定的名字加入下列默认资源：

- PACKAGEINFO
- PACKAGEOPTIONS
- DVCLAL
- DESCRIPTION

10.2.1 PACKAGEINFO 资源

PACKAGEINFO 资源用于保存当前模块及其所引用包的信息[①]。在 *SysUtils.pas* 中，该资源的结构定义为 TPackageInfoHeader 类型(其中，Flags 位掩码如表 10-1 所示)：

```
// code from SysUtils.pas
 PPackageInfoHeader = ^TPackageInfoHeader;
 TPackageInfoHeader = packed record
   Flags: Cardinal;   // 当前模块的标志字(亦称包程序标志字)
   RequiresCount: Integer;
   {Requires: array[0..9999] of TPkgName;
   ContainsCount: Integer;
   {Contains: array[0..9999] of TUnitName;}
 end;
```

其中，Flags 是当前模块的标志字(亦称包程序标志字)，其位掩码如表 10-1 所示：

表 10-1 模块标志字 Flags 的位掩码

标　识	值	说　明
pfNeverBuild	$00000001	外部建立的包
pfDesignOnly	$00000002	设计期包
pfRunOnly	$00000004	运行期包
pfIgnoreDupUnits	$00000008	对包中的单元忽略重复性检测
pfModuleTypeMask	$C0000000	
pfExeModule	$00000000	模块是可执行程序
pfPackageModule	$40000000	模块是包
pfLibraryModule	$80000000	模块是DLL
pfProducerMask	$0C000000	
pfV3Produced	$00000000	使用Delphi3或者C++ Builder3编译的模块
pfProducerUndefined	$04000000	使用未知编译器编译的模块
pfBCB4Produced	$08000000	使用C++ Builder4以上版本编译的模块
pfDelphi4Produced	$0C000000	使用Delphi4以上版本编译的模块

① 不要被名字迷惑，PACKAGEINFO 并不是包专用的资源，对于 Delphi 所编译的每一个模块(.EXE/.DLL/.BPL)，编译器都会加入该资源。

Requires 和 Contains 是两个数组，用于存放所有当前包所请求和包含的单元列表名（不包含扩展名）。数组长度存放在 RequiresCount 和 ContainsCount 域中。

PACKAGEINFO 结构并不是操作系统所需要的，所以除了让第三方程序可以识别模块类型之外，*.DLL* 和 *.EXE* 中的 PACKAGEINFO 没有意义。包的载入例程 LoadPackage() 会读取进程所有模块（含当前模块在内）的 PACKAGEINFO 资源，并进行单元重复性检测。因此如果模块需要载入运行期包，或者模块是包，那么 PACKAGEINFO 资源是必需的。

10.2.2 PACKAGEOPTIONS 资源

PACKAGEOPTIONS 是 Delphi 编译的默认资源中，惟一未公开结构的。但这并不表明 PACKAGEOPTIONS 中包含了多少技术内幕。事实上，它不过是编译配置文件*.*CFG* 在资源中的一个备份[1]。

分析该资源，并与*.*CFG* 对比，可以知道 PACKAGEOPTIONS 资源是如下的一个结构：

```
TPackageOptions = package record
  Unknow_Tag  : Byte;     // 未知，但值总为$30
  Directive_A : Byte;     // 选项：Record field alignment，编译条件：$ALIGN，
                          // 值：(0,1,2,4,8)
  Directive_B : Boolean; // 选项：Complete boolean eval，编译条件：$BOOLEVAL
  Directive_C : Boolean; // 选项：Assertions，编译条件：$ASSERTIONS
  Directive_D : Boolean; // 选项：Debug information，编译条件：$DEBUGINFO
  Directive_E : Boolean; //  - 未用 -
  Directive_F : Boolean; //  - 未用 -
  Directive_G : Boolean; // 编译条件：$IMPORTEDDATA，Use imported data
                         // references，
  Directive_H : Boolean; // 选项：Huge strings，编译条件：$LONGSTRINGS
  Directive_I : Boolean; // 选项：I/O checking，编译条件：$IOCHECKS
  Directive_J : Boolean; // 选项：Assignable typed constants，编译条件：
                         // $WRITEABLECONST
  Directive_K : Boolean; //  - 未用 -
  Directive_L : Boolean; // 选项：Local symbols，编译条件：$LOCALSYMBOLS
  Directive_M : Boolean; // 编译条件：$TYPEINFO，Runtime type information
  Directive_N : Boolean; //  - 未用 -
  Directive_O : Boolean; // 选项：Optimization，编译条件：$OPTIMIZATION
  Directive_P : Boolean; // 选项：Open parameters，编译条件：$OPENSTRINGS
  Directive_Q : Boolean; // 选项：Overflow checking，编译条件：$OVERFLOWCHECKS
```

① 这里的*.*CFG* 表明与项目文件名相同的一个配置文件名。这与在 *.pas* 文件中的{$R *.dfm}的含义是一致的。

```
  Directive_R : Boolean; // 选项: Range checking, 编译条件: $RANGECHECKS
  Directive_S : Boolean; //  - 未用 -
  Directive_T : Boolean; // 选项: Typed @ operator, 编译条件: $TYPEDADDRESS
  Directive_U : Boolean; // 选项: Pentium-safe FDIV, 编译条件: $SAFEDIVIDE
  Directive_V : Boolean; // 选项: Strict var-strings, 编译条件:
                         // $VARSTRINGCHECKS
  Directive_W : Boolean; // 选项: Stack frames, 编译条件: $STACKFRAMES
  Directive_X : Boolean; // 选项: Extended syntax, 编译条件: $EXTENDEDSYNTAX
  Directive_Y : Byte;    // 选项: Reference info/Definitions only,
// 编译条件: $REFERENCEINFO, $REFERENCEINFO, 值: (0,1,2)

// 编译条件: $MINENUMSIZE, Minimum size of enum types,值: (1,2,4)
  Directive_Z : Byte;

 // 当值=$0A, 表明 "Include TD32 debug info" 选项选中
  DllAndExeOptions : byte;
  MemoryAllocationMax : DWORD;  // 编译条件: $MAXSTACKSIZE
  MemoryAllocationMin : DWORD;  // 编译条件: $MINSTACKSIZE

  //未知, 但使用DCC32编译, 该值为$00100000, 而IDE下则为$0
  Unknow_Compile : DWORD;
  ImageBaseAddress : DWORD;     // 编译条件: $IMAGEBASE

  SizeOfConditionals : Byte;
  Conditionals : array [1..SizeOfConditionals] of Char;

  SizeOfUnitAliases : Byte;
  UnitAliases : array [1..SizeOfUnitAliases] of Char;

  SizeOfDescription : Byte;
  Description : array [1..SizeOfDescription] of Char;  // end by $0000


  // 未知, 值总为$00, 可能是字符串的长度值, 或结构的结束标志
  Unknow_EndOfStruct : Byte;
end;
```

这是一个形式化的结构，不可以直接在代码中使用。其中有少数几个域还是无法确定的，毕竟这是一个未公开的结构。

除非试图从编译后的模块文件中恢复出*.CFG 文件，该资源没有什么实际用途。Delphi 的包载入例程和 IDE 实际上都不需要从 PACKAGEOPTIONS 资源中获取什么有价值的信息。

10.2.3　DVCLAL 资源

DVCLAL（Delphi Visual Component Library Access License）资源并不是仅在使用 VCL 的情况下才会包含在资源节中。它仅表示编译时 Delphi 所使用的 VCL 的版本，当然也就表明这时使用的 Delphi 版本。因此它最常见的用途就是用来辨别模块是否使用 Delphi 编译，以及用哪种版本的 Delphi 编译。

任何一个 DVCLAL 资源都只是一个十六进制数据块（RC_DATA 类型的资源）。不同版本类型的 Delphi 的 DVCLAL 并不一样。下面是一些具体版本的值：

```
//for Delphi 3.xx Client/Server
A2 4C DF 96 7B 3C 3A 79 26 72 1F 09 0F 2A 25 17

//for Delphi 6,,7 Enterprise
26 3D 4F 38 C2 92 27 B8 E3 24 42 03 17 98 3A 83
```

可见，通过 DVCLAL 并不能确知是使用哪一个 Delphi 版本号。仅版本号而言，通过它能够获知的信息并不比 PACKAGEINFO 资源来得多。[①]

在 *SysUtils.pas* 中提供了验证 DVCLAL 的例程：

```
// 取授权信息( Get Delphi Access License )
function GDAL: LongWord;

// 验证是否是 Delphi Client/Server 版编译的模块
procedure RCS;

// 验证是否是 Delphi Professional 版编译的模块
procedure RPR;
```

在例程 GDAL()中，DAL 是指一个四字节的授权码，也称为 AL。GDAL()在资源中取得十六字节的 DVCLAL，然后通过固定的算法转换成一个四字节的 AL 返回。而在例程 RPR()中，通过与常量数组的元素 AL1s[1]或 AL1s[2]比较，即可测知编译时是否使用 Professional 版本的 Delphi。

在例程 RCS()中，先调用 ALR()来取得一个指向 DVCLAL 的 Buffer 的指针，然后调用

① 仍有一些工具可以侦测模块编译时使用的 Delphi 版本号，但是这些工具并不是根据分析资源来侦测的。例如 DeDe，是通过在 CODE 节中查找 TControl 类信息的起始地址来识别 Delphi 版本的；对于使用运行库的模块，则识别其运行库文件名。因此，DeDe 并不能侦测控制台应用的 Delphi 编译版本。

例程AL1()和AL2()进行运算，并将结果与AL1s和AL2s数组中的指定元素比较，从而测知编译时是否使用Client/Server版本的Delphi。

无须更深地了解两个验证算法细节。如果真的需要使用DVCLAL来确认Delphi的编译版本，仅需要抄录下几个十六字节的资源数据块就可以了。

10.2.4 DESCRIPTION资源

按照约定，DESCRIPTION资源用于设定一个EXE模块的描述说明。在Delphi中，可以使用{$DESCRIPTION}来定义这个描述。大多数情况下，Delphi推荐使用版本信息设置，而不是使用DESCRIPTION。

只有一个例外。在编译一个包时，应当使用DESCRIPTION来定义一个描述说明。因为Delphi会把这个说明作为IDE的安装包对话框中的包名。这种情况下，如果没有DESCRIPTION，在安装包对话框中，将以包的全路径文件名代替。

10.3 其他

资源字符串与字符串常量有明显的相同之处：至少在代码中使用起来与常量很相似。资源字符串是放在*.rsrc*节中的，而字符串常量放在CODE节中，它们在运行期都不能改写。

在静态修改PE文件时，由于资源字符串是在运行期通过TResStringRec记录的Identifier域来查找并载入的，因此在资源节中如何修改它的地址和大小，都不会导致运行期错误；而字符串常量是在代码中用地址访问的，因此变更它在CODE节中的位置和大小都可能导致错误。

资源字符串与字符串常量的效率基本一致，但字符串常量效率相对略高一些。由于没有引用计数机制，因此在频繁使用这两种类型字符串的场合中，应该先定义字符串变量来预取。

Delphi自动处理的资源事实上还包括在VCL应用程序中的.DFM文件，以及COM应用中的TLB表等等。此外用户代码还可以通过{$R}编译指令自由地向可执行文件模块中添加资源。但这些都不是“默认资源”，也不是内核一级的资源访问约定。

第 11 章

突破 Delphi 在应用开发上的限制

有些事的确是 Delphi 所不能做的，比如你想用它来打蚊子。知道这一点并不容易，因为的确有人试图开发能够驱蚊的软件。

知道一个开发工具能或者不能够做什么，是非常重要的。这可以使项目最初少走很多弯路。假如项目要求接管操作系统底层的文件 IO，那么这可能需要写一个驱动程序来实现。而这时开发经理决定要用 Delphi 来做这件事，其结果会使得开发人员在三个月的不停当机和重装操作系统之后，决定向公司递交辞呈。

并不是说在 Delphi 中不能做这件事，而是几乎没有任何可供参考的资料。要知道，除了.*DLL*、.*BPL* 和.*EXE*，要在 Delphi 中开发其他的任何模块，都不是一件容易的事。

接下来读者将看到一些已经古老得被人遗忘的技术，以及一些根本没有任何 Delphi 开发者涉足过的领域。支持我完成这一章的惟一理由是：我用 Delphi。

11.1 开发 DOS 保护模式程序

11.1.1 Delphi 1.0 与 DOS 16 位保护模式程序

现在能够在网上查到的、最早能够在 DOS 上运行 Delphi 程序是这样写的[①]：

```
// 首先使用Borland Pascal 7.0保护模式编译下面的代码，生成 DosApp.exe 程序

{ 1. DosApp.pas }
```

① 如下的代码来自于 http://bbs.ee.ntu.edu.tw/boards/Programming/10/2/9.html，我无法追查到它更早的来源。基本上来说，这些出自 1995 年的代码是混合 BP 与 Delphi 进行 DOS 保护模式开发的最早示例。

```
program DosApp;
uses interfce;
begin
  ProgMain;
end.

{ 2. interfce.pas }
unit interfce;

interface
  procedure ProgMain;

implementation
  procedure ProgMain; external 'MainLib' index 1;

end.


// 然后可以在Delphi 1.0中写代码，编译生成写 MainLib.dll

{ MainLib.dpr }
library MainLib;

procedure DllMain; export;
begin
  // ...
end;

var
  OldExitProc : pointer;
procedure MyExitProc; far;
begin
  ExitProc := OldExitProc;
  close(input);
  close(output);
end;

begin
  OldExitProc := ExitProc;
  AssignFile(output, 'con');
  Rewrite(output);
  AssignFile(input, 'con');
  Reset(input);
end.
```

Delphi 源代码分析

这个示例利用了 BP7.0 可以使用(也可以编写)DLL 模块的特性，使得 Delphi 的代码被作为一个 DLL 模块链入到应用中。而 *DosApp.exe* 这个程序，仅仅相当于一个载入器。

这是一个 Delphi 1.0 时代的技术，BP7.0 能识别的仅是使用 Delphi 1.0 编译的在保护模式上运行的模块。而且在运行时，还需要 *RTM.EXE* 和 *DPMILOAD.EXE* 的支持，以使得 *DosApp.exe* 被载入到 16 位 DOS 保护模式环境中。

还有一些人则致力于研究如何使用 Delphi 1.0 的语法特性(例如类、异常等)来直接编写和编译 DOS 保护模式程序。这其中最早可见的一个项目叫做 *"Delphi Does DOS"*。这是 Kevin R. Smith 在 1995 年 3 月公开的一篇文档，它详细地描述了如何改造 Delphi 1.0 和 Borland Pascal 来支持在 DOS 应用程序中使用 Delphi 1.0 的一些语言特性：

- 改写 RTL 源代码，使用 *DCC.EXE* 未公开的编译指令 "-CD" 来生成 DOS 实模式的 RTL .OBJ 模块。然后通过改造过的 DLIB 库管理程序来创建在 DOS 实模式下的、包含 Delphi 1.0 System 单元的库文件 *TURBO.TPL*。
- 使用 *DCC.EXE* 未公开的编译指令 "-CP" 来生成保护模式的 Delphi 1.0 System 单元 *SYSTEM.TPP*，并用 DLIB 将它包含到保护模式下的库文件 *TPP.TPL* 中。
- 改造 *SysUtils.pas* 和 *Classes.pas* 单元，使之可以使用于 DOS 保护模式程序中。

这样就可以使用 Delphi 1.0 的语法和一些基础类(例如 TList、TStream 等)来开发 DOS 实模式和保护模式程序了。

后来，在 1996 年至 1997 年间，Cosmin Truta（cosmint@cs.ubbcluj.ro）基于 Delphi Does DOS 项目的研究成果，发布了一个非官方的 Borland Pascal 8.0。在这个基于 BP 7.0 的升级版本上，可以用 Delphi 1.0 编译字符模式的 DOS 程序（保护模式和实模式），也可以使用 Turbo Vision。

16 位的 DOS 时代，终于彻底地结束了：字符模式终结于 Turbo Vision 和一个非官方的 Borland Pascal 8.0，图形模式则终结于 Delphi 1.0，而使用 BGI（Borland Graphics Interface）的程序则再也找不到了[1]。

11.1.2 WDOSX、Delphi 2.0–4.0 与 DOS 32 位保护模式程序

自 Delphi 2.0 开始，Delphi 的主程序文件名被更名为 *DELPHI32.EXE*，而命令行编译器被

[1] 在 DOS 保护模式下的另外一个著名的 Pascal 图形用户界面开发框架是 Graphic Vision(http://sourceforge.net/projects/gvision/)。它基于 16 位 DPMI，最大支持 1600*1200*256 色模式。但它使用的不是 BGI。

更名为 *DCC32.EXE*。这暗示了一个新的、32 位开发的时代的来临，这一切开始于 1996 年[1]。

Delphi 中可以编写控制台模式的应用程序，但这并不是 DOS 应用程序。Delphi32 与 DOS 程序之间惟一可能存在联系的是：32 位代码。

Delphi32 编译生成 32 位代码，因此这也意味着可能在 DOS 的 32 位保护模式（DPMI32）中运行这些代码。问题仅在于：谁来将 CPU 切换到 32 位保护模式并初始化这些代码的运行环境？

WDosX（Wuschel's Dos eXtender）用来解决这个问题[2]。

WDosX 使用一个名为 *stubit.exe* 的程序来为 Delphi 输出的控制台程序（PE 文件）绑定一个文件头。*stubit.exe* 实际上由三个独立的模块构成[3]：

- *WDOSX.DX*
- *PESTUB.EXE*
- *WDOSXLE.EXE*

stubit.exe 自动识别并选择合适的可执行文件头并绑定到目标文件，用以将 CPU 切换到 32 位保护模式环境中。除此之外，它分析目标文件的导入表，以决定将哪些模块包含到转换后的文件中。这些可动态嵌入的模块包括：

- *KERNEL32.WDL*
- *USER32.WDL*
- *OLEAUT32.WDL*
- *ADVAPI32.WDL*

这些模块为控制台应用初始化一个运行环境，模拟 Win32 的操作系统内核。它们是标准的 Win32 PE 文件（事实上它们是标准的 DLL 模块），提供一组 API 用来替代 Win32 操作系统内核中的对应模块（**.DLL*）的 API。例如：`CreateProcessA`, `LoadLibraryA`, `RegCloseKey`, `RegOpenKeyA`, `RegOpenKeyExA` 等。

WDosX 的使用非常简单：

- 在 Delphi 中编写一个控制台应用程序并编译（例 *DemoConsole.exe*）；
- 在 WDosX 的 *BIN* 目录中使用命令行为可执行程序添加文件头：

① 对于我个人而言，这是值得纪念的一年：我离开学校并进入软件公司。在这一年里我开始使用 Delphi 1.0，并见到在技术方面对我有深刻影响的 Danny.Chou，以及我后来的(而不是现在的)经理 P&J。

② WDosX 并不是惟一的选择，但对于 Delphi 开发者来说，它是最好的选择。这与后文将提到的另一个开源项目 "DWPL" 有关。

③ 使用 *stubit.exe* 的 "`-extract`" 参数可以分离出这三个文件。

```
C:\WDOSX\BIN>stubit DemoConsole.exe
```

这样，转换后的 *DemoConsole.exe* 就可以脱离 Windows 环境，直接在 DOS 命令行上运行了。

WDosX 不但包含了在 DPMI32 模式下的 CRT 和 DOS 单元（对于使用过 Turbo Pascal 或 Borland Pascal 的开发人员来说，一定知道这是什么），而且也提供了 VBE 单元来支持 VESA 接口的图形模式应用程序开发。此外，WDosX 还提供一个面向对象的 DPMI 接口单元 OODPMI，以允许 Delphi 程序员在 DPMI 环境中使用面向对象方法编写代码。

WDosX 不但支持 Delphi 编译的控制台应用程序[1]，也支持使用 TASM、NASM、Borland C/C++、MSVC++、Watcom C++和 DJGPP 等多种开发环境。此外，它为.EXE 加入的文件头是跨平台的，除了各种官方和非官方的 DOS 系统之外，还可以运行在 OS/2 和 Linux 的 DOS 仿真器中。

11.1.3 DWP、DWPL、Delphi 5.0–7.0 与 DOS 32 位保护模式程序中的图形接口

WDosX 已经做好了一切底层的工作，现在 Delphi 的开发人员可以编写控制台应用程序，然后通过 *stubit.exe* 来使它们在纯 DOS 环境中可以运行。

除了能使用 DOS、CRT 等单元、如同在 Turbo Pascal 中一样书写代码之外，WDosX 没有提供任何能够利用既有的 Delphi 代码的方法[2]。因此一个名为 DWP（Delphi WDosX Project）的开源项目组产生了。这个项目组致力于提供一个与 Delphi 类同的 RTL 环境，使得开发人员们能用 OOP 的方法来编写事件驱动的大型工程。而且这样开发的代码，仍然能够经过 WDosX 的 *stubit.exe* 转换，运行在纯 DOS 环境或其他操作系统平台中。

DWP 为此改写了包括 *System.pas* 在内的全部$(SRC)\RTL\Sys 目录下的全部代码，从而实现了 WDosX Core 库的一个增强版本（见表 11-1）。

然而仍然有些开发人员感到不能满足：用一个 RAD 的开发环境去做字符界面应用程序的确显得非常蹩脚。而这正是 DWPL（Delphi WDosX Project Library）这个开源项目所关注的问题[3]：

① WDosX 并不是专门为 Delphi/Pascal 而写的 DOS 32 位扩展器，它的代码基本上是用 TASM 编写的，但 Michael Tippach 在 WDosX 发布的源代码包中写上了一行“Welcome to Turbo Pascal 10!!”，我想这是对那个 Pascal 时代的向往和致敬。

② 对如何在 Delphi 中使用 WDosX 介绍得比较完整的文档是 László Agócsp 发布的“DOS development with WDOSX and Delphi”：http://dwp42.org/dwp/whitepapers/dwdx/index.htm。

③ DWP 与 DWPL 其实是一个项目中的两个部分。更确切地说，DWP 是 DWPL 的“Cross Platform Core Library”。DWP 有着比 DWPL 更早的起源，一些早期版本的 DWPL 中对项目名称的释义是“DWP Library”。

表 11-1 WDosX Core 库增强版的特性

特 性	提供	备 注
Turbo Pascal DOS&CRT Unit Emluator	WDosX	DWPL提供重新实现
VBE interface Unit	WDosX	DWPL扩展至支持VBE 1/2/3 interface
Calling real mode interrupt services	WDosX	
Debugger: WUDEBUG.EXE	WDosX	DWPL支持Delphi IDE调试器
DPMI Interface Unit	WDosX	DWPL提供增强支持
TCP/IP Stack for native DOS Packet drivers	DWP	
COM Device Access	DWP	
Thread and System timer	DWP	对多线程支持不完善
Long FileName Support	DWP	
TCP/UDP Support	DWP	包含对更多的网络协议的支持
Keyboard and mouse events	DWP	实现了非常底层的回调(Callback)
More Win32 Api Call	DWP	for Windows only

图 11.1 展示了 DWPL 项目组目前的成果：图的上半部分是在 Delphi IDE 中开发时的情况，下半部分则是点击 RUN 按钮之后在 DOS 窗口(以及纯 DOS)中的运行效果。



图 11.1 DWPL 项目组目前的成果

的确，通过 DWPL 就可以在 Delphi 环境中用 RAD 的方式开发 GUI 应用程序，然后通过特殊的编译选项（用于包含指定的库文件），将目标文件编译成在 DMPI32 模式上运行的字符界面应用程序。这个字符界面可以默认使用 Turbo Vision 风格的库，或者加挂第三方的开发包（例如图形模式的 AlpGraph 和将来版本的 Graphic Vision）。

DWPL 的目标显得与 WDosX 有些不同。WDosX 的目标仅是实现一个底层的、能够使 Delphi 的目标代码运行在 DPMI32 模式之上的扩展器，而 DWPL 试图实现一套字符界面的组件，并使它与图形界面上的组件对应起来。

DWPL 主要由这几个部分组成：

- BaseCLX：为 CLX 提供的基础平台，除了 DWP RTL 之外，还包括流（Stream）系统、RTTI 等。
- VisualCLX：目前开发中的版本是 WCLX2.0（（WDosX CLX library version 2.0）。
- NetCLX：目前使提供了简单的 WinSocket 支持。

WCLX2.0 基于 FreeCLX[1] 项目开发。这也意味着在 Delphi IDE 中使用 DWPL 设计图形界面时，应该创建 CLX Application，而不是普通的 Application。WCLX2.0 在运行期所使用的 Turbo Vision 字符型界面库则是来自于 Virtual Pascal 项目的一个修正版本。

11.2 开发 Windows9x 上的驱动程序

11.2.1 Delphi 编译的.OBJ 文件

WDosX 采用重写装载代码，并模拟 Win32 运行环境库的方法来使 Delphi 编译的在 32 位代码能够运行在 DOS 环境上。这是一个不错的方案。然而这对于“使用 Delphi 来开发 Windows9x 上的驱动程序 VxD[2]”这样的需求来说，却并不实用。

Delphi 能够编译适用于 VxD 的二进制目标代码，但是却不能直接生成 VxD。这是一切问题

① FreeCLX 是 SourceForge 的开源项目。一条来自 Borland 官方的信息是这样的：
http://community.borland.com/article/0,1410,27100,00.html

② VxD(Virtual x Device, Virtual Device Driver)用来虚拟化某个设备，使得每个进程都自认为拥有专属的该设备。关于 VxD 技术，一本非常好的书是侯捷先生译的《System Programming for Windows 95》，中文版名为《Windows 95 系统程序设计-虚拟机器与 VxD 程序设计》。可在侯捷先生个人网站(http://jjhou.csdn.net/)下载该书电子版。

的短结。然而 Borland 之所以在 Delphi 中不能编译生成 VxD 格式的文件，是因为 VxD 使用了一种非常古老的可执行文件格式 LE（Linear Executable），它可以同时包含 16 位和 32 位代码。虽然这正是运行在 Windows 9x 中的 VxD 所需要的（因为有些 VxD 需要包含实模式的初始化段），然而却是以纯 32 位的开发环境为发展方向的 Delphi 2 以上版本所无法提供的。

对于这种（无法输出指定格式的可执行文件）情况，Delphi 所能提供的仅是能够由链接器使用的 OBJ 文件。在 Delphi IDE 中选择"*Project\Options\Linker*"并选中"*Generate C object files*"（或在命令行上使用 *DCC32.EXE* 时加入"-J"参数），编译器就会在为每个单元生成 *.DCU* 文件的同时，生成一个 *.OBJ* 文件。

11.2.2 第三方的链接器处理 Delphi 的.OBJ 文件的问题

接下来的问题是如何使用这些 *.OBJ* 文件。由于 Delphi 并不能生成 VxD 格式的模块，所以只能用第三方的链接器来链接这些 *.OBJ* 并输出 *.VXD*。

然而 Delphi 的编译器为每一个独立的 *.OBJ* 生成了（至少）两个外部定义的（EXTDEF）符号：

- `@initialization`：当前单元的初始化节的指针；
- `@@HandleFinally`：`HandleFinally()`句柄指针。

这很容易理解。因为编译器认为在 Delphi 的内核中至少会包含这两个例程。但在使用第三方链接器时，并不知道有 Delphi 内核的存在。因此链接这些 *.OBJ* 文件时首先遇到的问题就是需要一个输出这些符号的目标文件。

如果使用"-J"参数编译 Delphi 内核，就可以得到 *System.obj* 和 *SysInit.obj*。在这种情况下，可以链接这两个目标文件中的`@initialization`和`@@HandleFinally`符号。这样做另外的好处是，可以使用这些单元中的一些内部例程。

如果不想重新编译内核，也可以用其他语言来编译输出这两个符号的 *.OBJ* 文件——在 Delphi 的单元中，是无法定义出这两个导出符号的。

用于输出这两个符号的一段非常简单的汇编代码可以是这样：

```
TITLE  Delphi Stub for VxD
.386p

_LTEXT   segment dword public 'LCODE'
         assume cs:_LTEXT

Public  @@HandleFinally
Public  @Initialization
```

Delphi 源代码分析

```
@@HandleFinally:
@initialization:
  ret

_LTEXT   ends

END
```

11.2.3 在 Delphi 中编写 VxD 驱动程序代码

使用 Delphi 来编写 VxD 驱动程序的技术方案[①]最早出自 Emil Biserov (fatty@mail.primorye.ru)发布的一份俄文技术文档。Riceball (teditor@mailroom.com) 将它译成了中文，这份文档已经在网络上流传得很广泛了。然而这份文档中并没有多少代码是用 Delphi 完成的，因此可移植性并不强。

其实除了在上一小节中所述的代码之外，其他所有有关VxD的代码都可以在Delphi中完成，而无须使用汇编代码。不过需要注意的是：绝大多数 User32 和 Kernal32 中的例程在 VxD 中都是无法调用的。因此也不应当在代码中引用 *windows.pas* 等单元。

一个框架性的 VxD 的 Delphi 程序可以是这样的：

```
unit vxdmain;

interface

// Device Descripter Block(DDB)
procedure DELPHIIO_DDB; stdcall;

implementation

function Control_0(R_EAX: Integer): LongBool; forward;
procedure DELPHIIO_DDB; stdcall;
asm
// ...
  DB      'DELPHIIO'
  DD      80000000h
  DD      OFFSET Control_0
// ...
```

① 你可以在本书的光盘上看到相关资料。需要留意的是：有且仅有 Delphi2-3 生成的 OMF 格式的目标文件(*.OBJ*)，能被 Microsoft ILink 转换为 COFF 格式并链接。因此后文中提到的代码仅能在这些版本中编译。

```
end;

function Control_0(R_EAX: Integer): LongBool;
begin
  // 检测参数dwControlMessage: R_EAX, 并决定调用相应的代码
  // ...

  // 置返回值,标志位
  asm
    mov @Result, eax
    cmp eax, 1
  end;
end;

end.
```

当操作系统将一个控制消息传给.VxD 驱动时，总是从模块导出的 DDB 块中去查找入口地址“OFFSET Control_0”，然后跳入相应的代码执行。操作系统由 EAX 寄存器传递给 VxD 的 ControlMessage 标识，在 Control_0()中，应当根据这个值来决定如何作进一步处理。

在编写自己的 VxD 时，上述框架中有两处是须要注意的：

- 在 DELPHIIO_DDB()中定义驱动的名字“DELPHIIO”时，可以改作其他的名字。但 VxD 的名字最多 8 个字符，并且它必须是大写！在操作系统全局范围内，所有 VxD 程序的名字不能重复。
- 导出例程 DELPHIIO_DDB()的名字可以更改。通常情况下，它应该是 VxD 名字并加上“_DDB”字符串后缀的格式。此外，当该例程名被修改后，在后文中将提到的一个 *vxddef.def* 文件中的相应定义也必须同时修改。

11.2.4　链接目标文件并输出 VxD 驱动程序

能够编译 LE 格式的 VxD 模块文件的链接器目前只有剩下三种：

- ILINK for DDK：VC2.0&MFC3.0 时代出现的 Incremental Linker。在 Windows 9x 的 DDK 包中包含一个可生成 VxD 的版本；
- LINK386：来自于早期的 OS/2 2.0 beta；
- WLINK：在 Watcom C/C++中包含的链接器。

选用 ILINK 链接 VxD 时，可以使用较为固定的命令行参数：

```
// 生成VxdStub.obj，参见前文所述，可以使用TASM编译。
C:\>ml -coff -DBLD_COFF -DIS_32 -W2 -c -Cx -Zm -DMASM6 VxdStub.asm
```

```
// 在Delphi中，通过项目文件生成全部的 .OBJ 文件
C:\>dcc32 -J -B -O- -Udelphi3 VxDProj.dpr

// 链接所有的目标文件(...)，并生成.VxD文件(xxx.vxd)
C:\>link /VXD /FORCE:MULTIPLE /DEF:vxddef.def /OUT:(xxx.vxd) (...)
```

这些参数中需要注意的是 link 程序的“/FORCE:MULTIPLE”参数。这是由于 Delphi 所生成的*.OBJ* 文件都将公布一个名为 Finalization 的结束化例程地址，这使得链接器会找到多个同名的符号。而“/FORCE:MULTIPLE”参数用于忽略这个错误。

“/DEF:vxddef.def”是必须的，它决定了链接器如何构造 VxD 文件的各个段，并且它将导出 VxD 驱动所必须的 DDB 块首地址。在 Delphi 中，这个 DDB 块将用一个例程的内嵌汇编代码来定义，例程的名字与 *vxddef.def* 文件中定义的必须一致。

在随书光盘中可以找到实现 VxD 驱动程序的一个项目的完整代码[①]。

11.3 开发 WinNT/2000 上的驱动程序

Delphi 很显然是一个纯 32 位开发环境的典型成功案例。Borland 为此做出了一些让步，例如不提供 LE 格式编译的目标模块。

许多人都应该知道，在 Windows NT/XP/2000 等操作系统中载入的 WDM（Windows Driver Model）驱动程序是纯 32 位代码、PE 格式的文件（扩展名为*.SYS*）。很显然，这与 Delphi 所编译生成的目标文件并没有任何的冲突。

在 Delphi32 不支持编译生成这样的目标文件的诸多（猜测的）理由中，有三条是更接近事实的：

- Delphi32 更注重面向普通开发人员的应用程序编写，而非特定人群的驱动程序编写；
- Delphi32 的内核代码无法安全地支持 WDM；
- Delphi32 无法提供一个安全的集成调试环境来支持 WDM 的开发，一个非安全的开发环境将会成为 Delphi 的市场壁垒。

① 这个项目是基于 Emil Viserov 所公布的那份俄文文档的，我用 Delphi 语言重写了多数的汇编代码，并完成了一个能更方便使用的项目框架，但整个项目的原型并没有实质性的改变。

如果能抛开来自市场的困扰[1]，要使 Delphi 可以开发.*SYS* 其实是非常简单的。因为 Nico Bendlin 的 MiniDExe 已经涤净了内核中的所有代码，而.*SYS* 在文件结构上与 Delphi 编译的 PE 格式的文件在物理结构中并没有什么不同。

使用.*EXE*、.*DLL* 或者.*BPL* 模块的内核最小化版本都可以编译出标准的.*SYS*。[2]但 Delphi 编译器所生成的模块初始化代码，将使原始参数的获取变得非常复杂。因此重写一个编译 WDM 格式驱动时专用的内核是非常必要的。

一个在编译 WDM 时可通用的.*EXE* 模块内最小化版本可以是这样的：

```
unit System;

interface

// ...
procedure _InitExe;

implementation

{ 编译器生成的模块初始化例程
  push ebp
  mov ebp,esp
  add esp,-$10
  mov eax,$00b91f5c      // 传入单元初始化表地址
  call @InitExe          // 调模块初始化例程
}
procedure _InitExe;
asm
  // 取本调用的返回地址
  pop edx

  // 修正堆栈
  add esp, $10

  // 取原始的ebp
  pop ebp

  // 在栈上压入返回地址
  push edx
end;
```

① 诸多的因素其实是起源于市场行为而并非技术方面的制约。做应用的开发人员应当从 Delphi32 的在技术上所采取的折中得到一些启发。这与本书着重于纯技术的讨论，却又并不鼓励在工程中实用这些技巧的初衷是一致的。

② 尽管.*BPL* 没有初始化代码，但它会多余地导出很多的内部例程和符号地址。因此并不是太好的选择。

```
//. ...
end.
```

在使用该内核代码时，可以使用类似下面这样的框架来完成 WDM 的编写：

```
program SampleWDM;
{$E sys}

uses
  winddk;

function _DriverEntry(DriverObject: PDRIVER_OBJECT;
  RegistryPath: PUNICODE_STRING): NTSTATUS; stdcall;
begin
  // 驱动程序的入口代码
  // ...

  // 返回SUCCESS
  Result := STATUS_SUCCESS;
end;

asm
{ ->    EAX Initable            }
{       [EBP+4] DriverObject    }
{       [EBP+8] RegistryPath    }
  // here, is Pure DriverEntry()
  jmp _DriverEntry
end.
```

在这个项目中必须引用一个 winddk 单元，这个单元可以从 Windows DDK 包中的 *winddk.h* 头文件中转换。[①]它定义了驱动程序开发中所需要的操作系统内核例程和常量。在项目中引用其他的单元（例如 *windows.pas*）是危险的，这与开发 VxD 驱动一样：驱动程序运行在内核模式，因此绝大多数 Win32 API 都无法安全地调用。

接下来需要修正 PE 文件头中的导入表。由于驱动程序载入例程需要使用导入描述表中的 `OriginalFirstThunk` 域值，而在 Delphi 编译的 PE 文件头中，该值总是为 0。因此需要设定一个有效的 `OriginalFirstThunk` 值，这可以使用 `FirstThunk` 域值来替代。光盘附带的工具 `WDMFixer` 可以完成这项修改：

① 本书光盘中包含了一个实现 WDM 驱动程序的完整示例，它改写自罗聪先生用 VC 实现的 HelloWDM(Hello World for WDM, http://www.luocong.com/)。该示例中所使用的 *winddk.pas* 并没有完全转换 *winddk.h*，我只转换了很少的一部分 ——完全转换的工作量是很难想像的。

```
C:\>WDMFixer SamlpeWDM.sys
```

在做最后一个操作之前，这样得到的.SYS 仍然是无法被操作系统所识别的。这源自于操作系统对驱动程序的一项限制：它将校验 WDM 驱动模块的 PE 结构头的 CheckSum 域的有效性，而 Delphi 所生成的 DLL 文件的该域总是 0 值。

因此，使 Delphi 编写的 WDM 能成功运行的最后一个关键是修正这个域。这用一些第三方的工具就很容易做到。当然，也可以使用 Microsoft 的 Link 程序：

```
C:\>LINK -EDIT /RELEASE /SUBSYSTEM:NATIVE SamlpeWDM.sys
```

11.4 其他

我并不赞同开发人员用 Delphi 去编写纯 DOS 应用或者驱动程序，因为这并不是最佳方案。在本书中讨论这些技术的原因，仅是因为前面所提到的生成三种文件格式的技术，是扩展 Delphi 开发能力的三种不同方法。基于这样的一些思想，是可以来实现 Delphi For Mobile，或者 Delphi For .NET CLR 的。

要点并不是 Delphi 能不能做到，而是开发人员对 Delphi 以及将要实现的目标了解多少。据我所知，国内已经有公司用 Delphi 完成了生成在.NET 上运行的中间代码编译器；而国外的一些独立组织，也在用 Delphi 开发虚拟机和操作系统。

我并不是开发驱动程序的专家。我也不能够指导读者去完成一个更有价值的驱动程序。我只是在本书中分析了在 Delphi 中做这些事的基本技术：编译指定格式的模块。

术语表

本书中各章节所使用到的术语，通常在第一次出现该术语时使用脚注或者行中的备注来解释。因此本术语表仅列出全书中一些通用术语的使用方法和概念。此外，也列出一些在其它语言(例如 C 语言)中存在差异的术语，以供参考。

本术语表并非官方(或直接取自官方文档的)解释。

1. 官方文档

通常是指 Delphi 正版产品中的使用手册，也包括在 Borland 官方网站上正式发布的一些技术文档。

2. 例程(Routines)

函数(Functions)和过程(Procedures)统称例程。

3. 上下文(Context)

Context 的直译。与一个目标对象或者具体事务相关的内存数据块，通常是一个记录。例如在操作系统中线程(对象)和进程(对象)都各有一个相关的上下文，而在 *System.pas* 中 `InitContext` 则是一个与模块初始化(事务)相关的上下文。

4. 声明(Declarations)与定义(Declares, Declaring)

常量、变量、类型以及其它所有 Delphi 中使用到的标识符(Identifiers)都需要声明后才能使用。声明的过程或者结果也被称为定义。本书中不区别这两个词的使用和语义。

5. 编译指示字(Directives)

通常也称为编译器指示字或编译指令，有编译开关、编译选项和编译条件三种类型。在 Delphi 中，编译开关和编译选项可以在命令行参数或者“Project Options”中设定，绝大多数也可以使用编译指示字在源代码中设定，例如“`{$APPTYPE GUI}`”和“`{$A+}`”。

条件编译指示字用于在源代码中设定编译的条件或者分支，例如“{$IFOPT A+}”；

6. 常量(Constants)

用 const 关键字定义，亦称常数。

7. 真常量(True constants)

相对于“类型化常量”而言的，普通意义上的常量。亦称真常数。

8. 类型化常量(Typed constants)

使用“Name : Type = Data”格式、用 const 关键字定义的常量。C/C++中，被称为静态变量。

9. 变量(Variables)

用 var 关键字定义。变量与变体(Variants)不是同一概念，后者是数据类型的一种(在本书中未讨论)。

10. 线程变量(Thread variables, Thread-local variables)

使用 threadvar 关键字定义。

11. 有初值变量(Initializing variables)

使用"Name : Type = Data"格式、用 var 关键字定义的变量。

12. 全局变量(Global variables)

在单元的 Interface 节中定义的变量。系统(进程装载器)会将全局变量置初值为 0。

13. 局部变量(Local variables)

在函数 / 过程 / 方法内部定义的变量。

14. 单元局部变量

在单元的 Implementation 节中定义的变量。系统(进程装载器)会将单元局部变量置初值为 0。

15. 系统内核单元

在本书中，通常指 *System.pas* 和 *SysInit.pas* 单元，某些情况下，也包括 *SysUtils.pas* 和 *Classes.pas* 单元。

16. 系统内部变量

在本书中，通常指在系统内核单元中声明、但未被公开的变量(单元局部变量)。

17. 系统全局变量

在本书中，通常指在系统内核单元中声明并公开的变量(全局变量)。

18. 写复制(copy-on-write)

读取一个内存区时不会有任何额外的代码产生，但写该内存区时，会导致一个新的、相同的内存区产生，并使写操作作用于新内存区的技术。

19. 值参数(Value parameters)

也称为传值参数或值参。在例程入口中默认声明的参数都是值参。传入变量的一个复制。

20. 变量参数(Variable parameters)

也称为变参。在例程入口中，以 `var` 关键字声明的参数都是变参。传入变量地址。

21. 常数参数(Constant (const) parameters)

以 `const` 关键字在例程入口声明。传入变量地址。

22. 输出参数(Out parameters)

是指用来输出结果值的参数。它以 `out` 关键字在例程入口声明。传入变量地址。

23. 无类型参数(untyped parameters)

使用 `const`、`out` 或 `var` 关键字定义，但不声明参数的类型。

24. 图形用户界面(GUI，Graphical UI(User Interface) application)

设计过程中是指图形界面设计，在 Delphi 中通常是指 GUI 界面的程序。

25. 控制台(CONSOLE，Console application)

在 Delphi 中，通常是指控制台界面的程序。本书部分行文中可能指 Windows 系统中的 DOS 窗口(DOS BOX，或命令提示符方式下的窗口)。

26. 弱连接

加入{$WEAKPACKAGEUNIT ON}条件编译的单元称为弱连接单元，包中包含(Contains)该单元的方式为弱连接。一些书籍中译为“虚包装(weakly packaged)”或“不紧密包装”。

参考书目

本书的撰写中，除行文的必要之外，会尽可能避免与现有书籍的重复，因此可能有部分的内容讲述得不够细致。读者可以阅读如下书籍(或电子文档)，以为弥补[1]。

参考书目中的电子文档可以通过网络下载，本书的随书光盘中并不包含这些电子文档。

《Object Pascal 语言手册》

《Object Pascal 语言手册》是 Borland 随产品售出的官方手册。目前为止，仅有 Delphi 1 和 Delphi 6 出过中文版本。这本手册对 Delphi 的语言特性和语法规则描述得非常完整，其中包含许多少为人知的细节。

《Object Pascal 语言手册》仅作为正版软件的组成部分提供。相对于 Delphi 在线帮助，该手册相当于“Delphi Language Reference”主题下的内容，该主题链接到($DELPHI)\Help\d7dlr.hlp 文件。

《Object Pascal 语言手册》可以作为本书如下章节的参考：

- 第 2 章：基本数据类型的实现
- 第 3 章：BASM(Boralnd 的汇编语言)精要
- 第 5 章：面向 Windows 开发的基本实现

[1] 《DELPHI 技术手册》和《Inside VCL 架构剖析》两本书也是非常重要的参考，但由于在“前言”中已经列出，故不再列。

《Object Pascal 参考》与《PASCAL 精要》

由于 Borland 官方的《Object Pascal 语言手册》基本来自 Delphi 的在线帮助，因此你可以直接查阅在线帮助来得到同样的信息。如果你不习惯阅读英文材料，那么你可以在网上得到一份由苏涌(cnsuyong@hotmail.com)先生以个人名义翻译和发布的中文电子文档：《Object Pascal 参考》。这份.CHM 格式的文档，基本完成了 Delphi 6 在线帮助中“Object Pascal Reference”主题的全部翻译。

与此相对应的，《PASCAL 精要》这份电子文档就并不直接翻译 Delphi 帮助系统。它由周良勇先生译自《Essential Pascal》。而《Essential Pascal》是 Marco Cantù[1] (http://www.marcocantu.com/)于 1999 年完成并以电子文档形式发布的。

基本上来说，《Object Pascal 参考》和《PASCAL 精要》都可以当作 Borland 官方的《Object Pascal 语言手册》的电子版本来阅读。

《WINDOWS 核心编程》

《WINDOWS 核心编程》译自 Jaffrey Richter 编写的《Programming Applications for Microsoft Windows》第四版，2000 年 5 月由机械工业出版社出版。这本书主要面向的是操作系统应用程序开发人员，以讲述操作系统的内部机制为主。该书使用 C 语言编写示例代码。

《WINDOWS 核心编程》可以作为本书如下章节的参考：

- 第 5 章：面向 Windows 开发的基本实现
- 第 7 章第 2 节：内存页管理
- 第 8 章第 3 节：Windows 与 Delphi 中的异常处理机制概要
- 第 9 章：多线程的实现及调度

(1) 《Delphi 入门到精通》及《Delphi 高级开发指南》两书的作者。两书的英文版书名分别是《Mastering Delphi x》和《Delphi Developer's Handbook》。

《Windows NT 技术内幕(第二版)》

David A. Solomon 的这本《Windows NT 技术内幕(第二版)》，是最具深度的介绍操作系统核心机制的书籍之一。至今为止我仅精读其中 1/4 或略多的内容，这常常令我感到遗憾。这本书的英文名为《Inside Windows NT, 2nd Edition》，中文版由北京博彦科技发展有限公司译，1999 年 7 月由清华大学出版社出版[1]。

《Windows NT 技术内幕(第二版)》可以作为本书如下章节的参考：

- 第 7 章第 2 节：内存页管理
- 第 9 章：多线程的实现及调度

《Windows 系统程序设计 大奥秘》

《Windows 系统程序设计 大奥秘》一书由侯捷先生译自 Matt Pietrek 编写的《Windows 95 System Programming SECRETS》。这本书并没有发行过简体中文版本。侯捷先生在其个人网站(http://jhou.csdn.net/)上公开了该书繁体版(PDF)的下载。

《Windows 系统程序设计 大奥秘》可以作为本书如下章节的参考：

- 第 6 章：Delphi 的积木艺术(PE)[2]
- 第 7 章第 2 节：内存页管理
- 第 8 章第 3 节：Windows 与 Delphi 中的异常处理机制概要
- 第 9 章：多线程的实现及调度

① 该书的第二版名为《Inside Microsoft Windows 2000, Third Edition》，中文版《Windows 2000 内部揭密》由简创锋等译，2001 年 10 月由机械工业出版社出版。但我建议读者从第二版开始阅读，而第二版仅作为理解操作系统版本差异时的少量参考。

② 关于 PE 文件结构的另一份不错的参考资料，是 Bernd Luevelsmeyer 于 1999 年编写的电子文档《The PE file format》(v1.9)，吴建佳(wuja@ms2.hinet.net)先生于 2002 年将它译成了繁体中文的电子版，名为《PE 档案格式》。

《Windows 95 系统程序设计-虚拟机器与 VxD 程序设计》

《Windows 95 系统程序设计-虚拟机器与 VxD 程序设计》一书由候捷先生译自 Walter Oney 编写的《System Programming for Windows 95》。这本书并没有发行过简体中文版本。候捷先生在其个人网站(http://jjhou.csdn.net/)上公开了该书繁体版(PDF)的下载。

《Windows 95 系统程序设计-虚拟机器与 VxD 程序设计》可以作为本书如下章节的参考：

- 第 11 章第 2 节：开发 Windows9x 上的驱动程序

《Microsoft Windows 驱动程序模型设计》

Walter Oney 编写的《Programming The Microsoft Windows Driver Model》一书并没有发行过中文版本。为此马少华(mhsinet@263.net)先生翻译了该书，驱动开发网(http://www.driverdevelop.com/)发布了该译作的电子版本。

《Microsoft Windows 驱动程序模型设计》可以作为本书如下章节的参考：

- 第 11 章第 3 节：开发 WinNT/2000 上的驱动程序

[General Information]
书名=Delphi源代码分析
作者=
页数=350
SS号=0
出版日期=

封面
书名
版权
前言
目录
正文